Ingeniería Técnica de Telecomunicación, Sistemas de Telecomunicación

Departamento de Ingeniería Telemática

PROYECTO FIN DE CARRERA

Estudio de rendimiento de conexiones TLS con ECC en dispositivos Android

Autor: Jesús Manuel Calvo Corrales

Tutora: Florina Almenares Mendoza

Directora: Patricia Arias Cabarcos

Leganés, 28 de octubre de 2015

Título: Estudio de rendimiento de conexiones TLS con ECC en dispositivos Android
Autor: Jesús Manuel Calvo Corrales
Tutor: Florina Almenares Mendoza
Director: Patricia Arias Cabarcos

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 28 de Octubre de 2015 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

Primero quiero agradecer a mi familia y en especial a mis padres Manoli y Paco, por su apoyo incondicional, sin ellos nada hubiera sido posible.

También quisiera agradecer a todos los que están o han estado acompañándome en este largo camino, en especial a Javier y Elena por su eterna paciencia.

Especial mención a mis tutoras Florina y Patricia por su compresión y ayuda durante la realización de este proyecto.

Y por último a dos cositas llamadas Ainoa y Ainara por ser una de mis alegrías durante este tiempo.

Resumen

La seguridad en las conexiones móviles es algo vital, en especial en los dispositivos con el sistema operativo Android al ser el de más implantación en el mercado. Los protocolos y algoritmos usados tradicionalmente empiezan a no responder a las necesidades de seguridad y eficiencia que este tipo de dispositivos necesita. Es por ello, por lo que la implementación de la criptografía de curva elíptica (ECC) puede ser fundamental para aumentar la seguridad y eficiencia de las conexiones en dispositivos Android.

Para estudiar la eficiencia de dichas conexiones se ha realizado la medida de diferentes parámetros de las conexiones seguras desde un Smartphone Android. Para ello, se ha implementado un entorno de pruebas configurable donde poder realizar las diferentes pruebas.

En primer lugar se ha medido la eficiencia de conexiones TLS mediante ECC entre los navegadores comerciales de Android más usados y un servidor que acepta conexiones seguras configurado para las pruebas.

En segundo lugar se han estudiado las diferencias entre certificados RSA y certificados ECDSA en las conexiones TLS entre el dispositivo Android y el servidor configurado. Para ello se ha desarrollado un navegador reducido usando las herramientas que Android ofrece para ello, permitiendo realizar la gestión de certificados y conexiones de la forma que se había diseñado para la prueba.

Finalmente, usando el navegador programado, se ha probado el soporte de los cifradores ECC de Android que ofrecen los servidores comerciales más visitados, comprobando si cumplen las recomendaciones NIST.

Con todo ello se ha podido llegar a diferentes conclusiones al respecto de la eficiencia del uso de ECC en las conexiones seguras en Android con respecto a otros algoritmos y protocolos, haciendo especial hincapié en las diferencias entre RSA y ECDSA.

Palabras clave: ECC, ECDSA, RSA, Android, TLS, Seguridad, Smartphone

Abstract

Security in mobile connections is very important, especially in devices with the operating system Android because it is one of the operating systems with a higher market penetration. The Protocols and algorithms used traditionally do not give answers to security and efficiency needs. This is the reason why the implementation of elliptic curve cryptography is critical to increase connections' security and efficiency in Android devices.

To study the efficiency in these connections, security parameters have been measured using an Android smartphone in a configurable test environment.

Firstly, TLS connection efficiency has been measured for a set of Android commercial browsers.

Secondly, the difference between RSA certificates and ECDSA certificates has been measured in TLS connections established from the Android device. To do this, a web browser has been developed using the Android cryptographic libraries.

Finally, we used the programmatic browser to test the ECC support offered by the ten most visited Internet web servers, and determine if they are compliant with NIST security recommendations.

The study ends with conclusions about both the efficiency and security features of ECC usage in Android, highlighting the differences between RSA y ECDSA.

Keywords: ECC, ECDSA, RSA, Android, TLS, Security, Smartphone

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

INTRODUCCIÓN

1.1 Motivación del proyecto

En la actualidad la mayor parte de las conexiones a Internet se realizan a través de dispositivos móviles en detrimento del uso de ordenadores. Sin embargo, la mayoría de las tecnologías usadas en ello son prácticamente las mismas que las usadas anteriormente a pesar de que los dispositivos móviles por sus características necesitan otras especificaciones más precisas para que sean eficientes.

Los móviles presentan una limitación clara con respecto a otros dispositivos que se conectan a Internet, y es la autonomía. Un móvil depende continuamente de la energía almacenada en sus baterías por lo que tanto su *hardware* como su *software* deben ser totalmente eficientes para reducir el consumo de energía al mínimo sin perjudicar las capacidades del dispositivo y así aumentar su autonomía. La optimización en energía es algo totalmente fundamental para que un dispositivo móvil sea útil, pues no tendría sentido su uso si tiene que estar continuamente conectado a la red ya que perdería su principal característica que es la movilidad.

De todo lo comentado anteriormente surge la motivación para realizar este proyecto. Estos dispositivos se usan principalmente para conectarse a Internet por lo que la seguridad en dichas conexiones es fundamental, sobre todo en estos dispositivos que están continuamente conectados y que albergan un gran número de datos personales del usuario.

Dichas conexiones seguras aportan una sobrecarga (pequeña o grande dependiendo de los casos) que en los dispositivos que estamos estudiando es fundamental para la eficiencia de dichas conexiones. Esta sobrecarga depende del tipo de protocolo usado en las conexiones seguras, por lo que la elección de un protocolo más eficiente puede ser fundamental para mejorar la eficiencia de los dispositivos móviles.

La mayoría de los dispositivos móviles usados funcionan mediante el sistema operativo móvil proporcionado por Google, es decir, Android. Es por ello que un estudio de la eficiencia en dispositivos móviles debe empezar por estudiarlo en este tipo de dispositivos.

Android ha heredado los protocolos y algoritmos de conexiones seguras usadas en los dispositivos y sistemas no móviles. Esto es debido a que los dispositivos Android deben ser capaces de conectarse con los mismos servidores que el resto de dispositivos si quieren ser útiles. Esto hace que herede algoritmos que para conexiones móviles sean poco eficientes produciendo grandes consumos de energía a medida que aumentamos la seguridad.

De todos los algoritmos heredados por Android hay un tipo que podría ser más eficiente que el resto para dispositivos móviles debido a sus características que hacen que necesiten menos recursos que el resto ofreciendo un nivel parecido de seguridad. Se trata de la criptografía de curva elíptica (ECC), un tipo de criptografía asimétrica cuyo estado de implementación y soporte actual es todavía inmaduro por ser más compleja y de diseño más reciente con respecto a otros algoritmos ya bien establecidos y probados.

Por todo esto, en este proyecto estudiaremos las conexiones seguras mediante ECC en dispositivos Android.

1.2 Objetivos

El objetivo principal de este proyecto consiste en realizar un estudio sobre la eficiencia del uso de ECC en conexiones seguras en dispositivos Android, estudiando la energía consumida y el tiempo empleado en las conexiones para diferentes configuraciones. También estudiaremos el soporte a los algoritmos ECC.

Nuestra meta será establecer si realmente es mayor la eficiencia de los algoritmos ECC en dispositivos Android y si esta eficiencia compensa en términos de tiempo y energía con respecto al uso de otros algoritmos.

Para ello, compararemos los valores obtenidos con los teóricos y con los obtenidos en otros trabajos anteriores [1].

En primer lugar estudiaremos el soporte y la eficiencia de las conexiones de diferentes navegadores comerciales de Android y compararemos los resultados con trabajos anteriores, de esta forma veremos cómo es el soporte de dichos navegadores a ECC y si su eficiencia aumenta con el uso de dicho algoritmo.

En segundo lugar estudiaremos el consumo de tiempo y energía durante el intercambio de certificados RSA y ECDSA. Dichas medidas se realizan específicamente en el proceso de *handshake* de TLS, con el objetivo de observar el consumo energético

y de tiempo justo durante dicho proceso. Para ello es necesario desarrollar un navegador en Android que realice las medidas sólo durante el *handshake* y que lo separe del intercambio de datos.

Por último, usando dicho navegador programático, estudiaremos el soporte de ECC en los principales servidores comerciales.

1.3 Estructura de la memoria

En este primer capítulo se ha descrito la motivación y los objetivos del proyecto. A continuación se incluye como se estructura el resto de la memoria junto con un breve resumen de cada capítulo, para facilitar la lectura de este documento.

Capítulo 2 Estado del Arte: Se describen los principales protocolos y algoritmos usados en conexiones seguras poniendo especial énfasis en los relacionados con la criptografía de curva elíptica (ECC). También se describe el funcionamiento de la plataforma Android, de los servidores HTTP Apache y de su módulo OpenSSL.

Capítulo 3 Metodología: Se plantea la metodología empleada para evaluar el tiempo empleado y la energía consumida en realizar conexiones seguras mediante ECC en dispositivos Android, exponiendo los objetivos a cumplir y justificando cómo se ha desarrollado.

Capítulo 4 Estudio de eficiencia en navegadores comerciales: En este capítulo se presentan las medidas de tiempo obtenidas en las pruebas con navegadores comerciales mediante la metodología descrita en el capítulo 3 y se validan realizando una comparación con los resultados obtenidos en otro estudio. Finalmente se sacan conclusiones de rendimiento de las conexiones seguras de navegadores comerciales de Android.

Capítulo 5 Estudio de eficiencia en navegador programático: En este capítulo se presenta las medidas de tiempo y energía obtenidas en las pruebas con el navegador programado mediante la metodología descrita en el capítulo 3. Con los resultados se realiza una comparación entre los certificados RSA y ECDSA. También se estudiará el soporte a los cifradores ECC implementados por Android en servidores comerciales.

Capítulo 6 Planificación y Presupuesto: Se detallan las fases del desarrollo del proyecto y se incluye una descripción de los diferentes costes para obtener el presupuesto total de proyecto.

Capítulo 7 Conclusiones y Trabajo Futuro: Por último, se exponen las conclusiones a las que se ha llegado tras la realización de este proyecto y se explica el trabajo futuro que se puede realizar para completar los resultados obtenidos.

Apéndice: Se detallan los valores resultantes de las pruebas de tiempo y energía realizadas incluyendo todas las repeticiones hechas de cada medida.

Capítulo 2

ESTADO DEL ARTE

En este capítulo se explicará paso a paso cómo funcionan los algoritmos y protocolos asimétricos usados en las conexiones TLS/SSL (*Transport Layer Security*/ *Secure Socket Layer*) para firma y validación utilizados durante la fase de *handshake*.

Se hará más hincapié en explicar los protocolos y algoritmos basados en curvas elípticas (ECC, *Elliptic Curve Cryptography*), en especial su uso en Android y los certificados ECDSA (*Elliptic Curve Digital Signature Algorithm*).

También se estudiará el funcionamiento de las conexiones TLS/SSL usando un servidor Web

2.1 Criptografía asimétrica

El uso generalizado de redes informáticas en el tratamiento y transmisión de la información, así como el aumento constante del número de usuarios de estos sistemas, han motivado la necesidad de mejorar la seguridad en las comunicaciones. Son muchas y variadas las situaciones donde cabe garantizar la confidencialidad, la integridad o la autenticación de la información transmitida. Tales necesidades se han podido satisfacer mediante el uso de distintos protocolos criptográficos, en los que se combinan a menudo criptosistemas de clave compartida (criptografía simétrica) con los llamados criptosistemas asimétricos o criptosistemas de clave pública. [2]

2.1.1 Descripción y funcionamiento

Uno de los inconvenientes que tiene la criptografía de clave compartida es la necesidad de un canal seguro para intercambio de la clave, ya que dos usuarios tienen de pactar previamente su clave secreta si desean cifrar los mensajes que se van a enviar. Para solventar este problema, Diffie y Hellman proponen en 1976 un intercambio seguro de claves entre dos usuarios, sin necesidad de depender de un canal seguro previamente. Éste es el primer paso que origina la criptografía de clave pública, en tanto que no va a ser necesaria la comunicación directa entre los usuarios. Bastará con emplear cierta información que el receptor hace pública y, por tanto, accesible para cualquier entidad que se quiera comunicar con él. La seguridad de un criptosistema de clave pública reside entonces en problemas matemáticos computacionalmente difíciles, es decir, problemas para los que no se conocen, a día de hoy, algoritmos eficientes para resolverlos.

Entre los criptosistemas de clave pública más usados actualmente podemos citar el sistema **RSA**, descrito por Rivest, Shamir y Adleman en 1977, basado en el problema de la factorización de enteros, y el criptosistema **El Gamal** basado en el problema del logaritmo discreto sobre el grupo multiplicativo de un cuerpo finito. Sin embargo, el avance en la eficiencia de los nuevos algoritmos ha provocado la necesidad de aumentar el tamaño de las claves usadas para garantizar la seguridad de las comunicaciones. Es por ello que en la actualidad ha aumentado el uso de ECC (*Elliptic Curve Cryptography*) que mediante el uso de algoritmos basados en curvas elípticas proporciona tamaños de clave menores para el mismo nivel de seguridad.

Las dos principales operaciones de la criptografía de clave pública son:

- **Cifrado de clave pública**: un mensaje cifrado con la clave pública de un destinatario no puede ser descifrado por nadie (incluyendo al que lo cifró), excepto un poseedor de la clave privada correspondiente. Este será

el propietario de esa clave y la persona asociada con la clave pública utilizada.

- **Firmas digitales**: un mensaje firmado con la clave privada del remitente puede ser verificado por cualquier persona que tenga acceso a la clave pública del remitente, lo que demuestra que el remitente tenía acceso a la clave privada (y por lo tanto, es probable que sea la persona asociada con la clave pública utilizada) y la parte del mensaje que no se ha manipulado.

- **Funcionamiento criptografía asimétrica o de clave pública**

El funcionamiento de un sistema criptográfico asimétrico o de clave pública es relativamente sencillo. Se utiliza una pareja de claves asociadas entre ellas, de las cuales una es pública, que debe ser conocida por todo el mundo, mientras que la otra es privada, sólo debe conocerla el propietario de dicha clave. Por tanto, para una operación de cifrado, una se usa para cifrar y la otra para descifrar, respectivamente. Conociendo la clave pública o el texto cifrado no se puede obtener la clave privada, por tanto, con la clave pública cualquiera puede cifrar un mensaje, pero sólo el propietario de clave privada puede descifrarlo. La Figura 2.1 muestra lo explicado anteriormente.

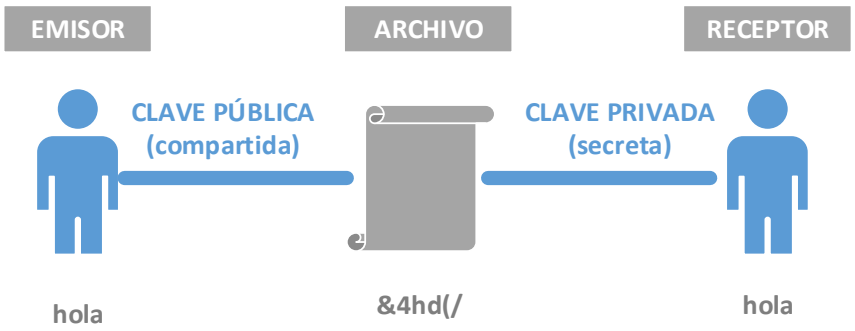


Figura 2.1 Esquema del funcionamiento básico criptografía asimétrica de clave pública

Por tanto, para realizar la distribución de claves secretas, el emisor envía la clave secreta, generada aleatoriamente, cifrada con la clave pública del receptor, el único capaz de descifrarla usando su correspondiente clave privada.

La autenticación se realiza mediante el cifrado del emisor con su clave privada, operación que sólo él puede realizar. Cualquiera puede descifrarlo con su clave pública, verificando así su autoría. El mensaje es comprimido antes de ser firmado.

Matemáticamente esto se implementa mediante el uso de funciones unidireccionales, es decir, funciones de las que se puede realizar su cálculo directo, pero calcular su función inversa es tan complejo que es casi imposible realizar, y ahí es donde radica la seguridad de este tipo de criptografía. Los problemas matemáticos que se usan para implementar esto pueden ser:

– Factorización: descomponer un número grande en sus factores primos.

– Logaritmo discreto: obtener el exponente al que ha sido elevado una base para dar un resultado.

– Mochila tramposa: obtener los sumandos que han dado origen a una suma.

- **Ventajas y desventajas**

La mayor ventaja de la criptografía asimétrica es que la distribución de claves es más fácil y segura ya que la clave que se distribuye es la pública manteniéndose la privada para el uso exclusivo del propietario. Sin embargo, posee varias desventajas:

- Para una misma longitud de clave y mensaje se necesita mayor tiempo de proceso que usando criptografía simétrica.
- Las claves deben ser de mayor tamaño que las simétricas. (Generalmente son cinco o más veces de mayor tamaño que las claves simétricas).
- El mensaje cifrado ocupa más espacio que el original.

Los nuevos sistemas de clave asimétrica basado en curvas elípticas tienen características menos costosas, solucionando o al menos mejorando algunas de las desventajas anteriores. Para llegar a un grado aceptable de seguridad, RSA y DSA deberían usar claves de 2048 bits, mientras que para la ECC sería suficiente con 224. A medida que la clave crece, aumenta la distancia entre la seguridad de cada propuesta. Por ejemplo, el ECC con 380 bits es mucho más seguro que RSA o DSA con 2000 bits (de hecho, para esta longitud de clave, el ECC es comparable al RSA de 7600 bits).

2.1.2 Algoritmos y protocolos

Algunos algoritmos y tecnologías de clave asimétrica son:

- **Diffie-Hellman**

El protocolo de cifrado Diffie-Hellman (recibe el nombre de sus creadores) es un sistema de intercambio de claves entre partes, que no han contactado previamente, a través de un canal inseguro y sin autenticación.

Este protocolo se utiliza principalmente para intercambiar claves simétricas de forma segura para posteriormente pasar a utilizar un cifrado simétrico, menos costoso que el asimétrico. Se parte de la idea de que dos interlocutores pueden generar de forma conjunta una clave sin que ésta sea comprometida. El proceso es el siguiente:

Cuando dos usuarios A y B quieren establecer una clave secreta común, el protocolo es el siguiente:

1. Se establecen un primo "p" y un generador $g \in Z^*_p$. Estos dos valores ("g" y "p") son públicos. Siendo Z^* el conjunto de los enteros menores que "p", que son primos relativos de éste y además es un grupo bajo la multiplicación módulo "p".
2. A escoge $x \in Z_{p-1}$ al azar, calcula $X = g^x \pmod p$, y envía X a B.
3. B escoge $y \in Z_{p-1}$ al azar, calcula $Y = g^y \pmod p$, y envía Y a A.
4. A calcula $K = (g^y \bmod p)^x \bmod p$
5. B calcula $K = (g^x \bmod p)^y \bmod p$
6. Siendo la clave "K"

Si alguien intercepta la conversación entre A y B, por ejemplo, un usuario malintencionado "E", que posee p, g, X e Y, podría calcular el secreto compartido si tuviera también uno de los valores privados (x o y) o lograse invertir la función, pero como se mencionó anteriormente estos algoritmos usan funciones trampa, ya que para calcular x dado X tenemos que el problema del logaritmo discreto en Zp* es un problema que se cree intratable.

No obstante, este protocolo es vulnerable al ataque de hombre en el medio ("*man-in-the-middle*"). Estos ataques consisten en que un tercero se coloca en medio del canal y hace creer a ambos que es el otro. De esta forma se podría acordar una clave con cada parte y servir de "enlace" entre los dos participantes. Para evitar esto se puede emplear un protocolo de autenticación de las partes mediante por ejemplo TLS.

- **RSA (Rivest, Shamir, Adleman)**

El algoritmo fue descrito en 1977 por **Ron Rivest, Adi Shamir y Len Adleman** en el MIT (*Massachusetts Institute of Technology*). Es un algoritmo de cifrado de clave pública (o asimétrica) por bloques, que sigue los siguientes pasos:

1. Se construye un número "N", que resulta del producto de dos primos "p" y "q" Teniendo : $N = p \cdot q$ y $\Phi(N) = (p-1) \cdot (q-1)$
2. Se selecciona un número "e", tal que, $1 < e < \Phi(N)$ MCD $(e, \Phi(N)) = 1$ ("e" y $\Phi(N)$ son primos relativos).
3. Se calcula el inverso de "e", denotado por "d", tal que $e \cdot d = 1 \pmod{\Phi(N)}$
4. Con esto se consiguen las claves (e, d), siendo la clave pública (e, N) y la clave privada (d, N).

   Una vez tenemos las claves podemos pasar a cifrar/descifrar los mensajes:

- Cifrado: $C = M^e \pmod N$ con MCD (M, N) = 1 y M < N
- Descifrado: $M = C^d \pmod N$

La seguridad de este algoritmo radica en que no hay maneras rápidas conocidas de factorizar un número grande en sus factores primos utilizando computadoras tradicionales. Utiliza claves de 512, 768, 1024 o 2048, siendo la más típica la de 1024 bits.

- **ElGamal**

Este algoritmo está basado en el algoritmo de Diffie-Hellman para el intercambio de claves, fue descrito en 1984 por Taher Elgamal. Actualmente se usa en el software libre GNU Privacy Guard (GPG), versiones de PGP y otros sistemas.

El algoritmo sigue los siguientes pasos:

1. El receptor selecciona un primo grande "p" y un generador $g \in Z^*_p$.
2. El receptor selecciona aleatoriamente un valor "x_r" tal que $0 < x_r < p-1$. Siendo "x_r" la clave privada.
3. La clave pública es igual a $y_r = g^x_r \pmod p$.
4. Ahora el emisor toma un "K" aleatorio tal que MCD (K, p-1) = 1, es decir, "K" sea primo relativo con p-1 y calcula $r = g^K \pmod p$ y $s = M \cdot y^K_r \pmod p$.
5. El emisor manda el mensaje cifrado como C = (r, s), siendo "r" la clave pública del emisor y "s" el mensaje cifrado.
6. Ahora el mensaje cifrado se descifra calculando $M = s/r^x_r$.

La seguridad del algoritmo depende en la dificultad de calcular logaritmos discretos.

- **DSA**

El algoritmo de firma digital (DSA, *Digital Signature Algorithm*) emplea un algoritmo de firma y cifrado distinto al de RSA, aunque ofrece el mismo nivel de seguridad. Lo propuso el *National Institute of Standards and Technology* (NIST) en 1991 y fue adoptado por los *Federal Information Processing Standards* (FIPS) en 1993. Desde entonces se ha revisado cuatro veces. Es el estándar del Gobierno Federal de los Estados Unidos de América.

Inicialmente se utilizaban claves de 512 bits y posteriormente se incrementó a 1024 bits para mayor seguridad.

Una desventaja de este algoritmo es que requiere mucho más tiempo de cómputo que RSA.

- **DSS**

Protocolo de firma digital basado en el DSA (digital signature algorithm), como estándar de firma digital. El DSS se basa en el criptosistema ElGamal.

El algoritmo sigue los siguientes pasos:

El usuario A quiere firmar un mensaje m:

- A escoge un número $k \in Z/p - \{0,1,p-1\}$ al azar, tal que $mcd(k,p-1) = 1$ y
- Calcula a^k,
- Calcula h(m), donde h(.) es una función hash,
- Calcula $s \in Z/(p-1)$ verificando, $h(m) = nA \cdot a^k + k \cdot s \pmod{(p-1)}$.
- La firma de m es la pareja (a^k,s).

Un usuario que quiera verificar la firma del mensaje m deberá hacer:

- Calcular el hash de m, h(m),
- Obtener del directorio público la clave pública de A: a^{nA} ,
- Validar la firma comprobando la siguiente igualdad: $a^{h(m)} = (a^{nA})^{ak}(a^k)^s \pmod{p}$.

- **TLS**

Es un protocolo criptográfico que proporciona comunicaciones seguras en una red. Usa criptografía asimétrica para autenticar a la contraparte con quien se están comunicando, y para intercambiar una llave simétrica. Esta sesión es luego usada para cifrar el flujo de datos entre las partes.

TLS proporciona autenticidad, confidencialidad e integridad de la información entre extremos sobre Internet mediante el uso de criptografía. Habitualmente, sólo el servidor es autenticado (es decir, se garantiza su identidad), mientras que el cliente se mantiene sin autenticar, pero permite la autenticación mutua.

TLS implica una serie de fases básicas:

- Negociar entre las partes el algoritmo que se usará en la comunicación.
- Intercambio de claves públicas y autenticación basada en certificados digitales.
- Cifrado del tráfico basado en cifrado simétrico.

Durante la primera fase, el cliente y el servidor negocian qué algoritmos criptográficos se van a usar. Las implementaciones actuales proporcionan las siguientes opciones:

- Para criptografía de clave pública: RSA, Diffie-Hellman, DSA (Digital Signature Algorithm) o Fortezza.
- Para cifrado simétrico: RC2, RC4, IDEA (International Data Encryption Algorithm), DES (Data Encryption Standard), Triple DES y AES (Advanced Encryption Standard).
- Con funciones hash: MD5 o de la familia SHA.

- **Criptografía de curva elíptica (ECC)**

La criptografía de curva elíptica la estudiaremos en detalle en los siguientes apartados de este proyecto por ser el foco principal del estudio.

2.2 Criptografía de curva elíptica (ECC)

En los últimos años, la criptografía con curvas elípticas ha adquirido una creciente importancia formando parte de los estándares industriales. Si bien se han diseñado variantes con curvas elípticas de criptosistemas clásicos, como el RSA, su principal logro se ha conseguido en los criptosistemas basados en el problema del logaritmo discreto, como los de tipo ElGamal. En este caso, los criptosistemas elípticos garantizan la misma seguridad que los construidos sobre el grupo multiplicativo de un cuerpo infinito primo, pero con longitudes de claves mucho menores.

A continuación veremos las propiedades de estos criptosistemas y los requerimientos básicos para que una curva sea criptográficamente útil. Además veremos las diferentes formas de implementación y el uso real que actualmente tiene.

2.2.1 Curvas elípticas

Se habla de curva elíptica en referencia a una ecuación $y^2=x^3+Ax+B$ que cumple $4A^3+27B^2\neq 0$. Dando diferentes valores a A y B obtenemos todo un conjunto de curvas que, al ser dibujadas, ofrecen una forma similar.

En la Figura 2.2 observamos dos ejemplos de curvas elípticas:

$y^2=x^3-x$ $y^2=x^3-x+1.$

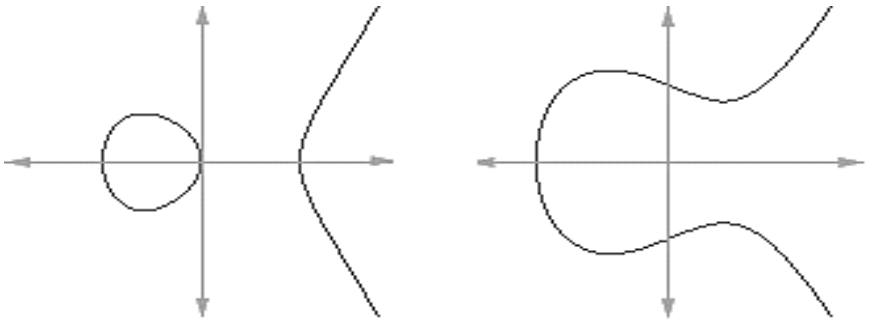


Figura 2.2. Ejemplos de curvas elípticas.

Si queremos dar una definición más formal de una curva elíptica podríamos decir: *"Una curva elíptica es una curva plana no singular de grado 3 junto con un punto racional prefijado, que denominaremos punto base"* [3].

Las curvas elípticas tienen ciertas características que las hacen especiales en el mundo de la criptografía. Una de estas características consiste en la posibilidad de poder generar un punto en una curva partiendo de dos puntos dados (o incluso de uno). Explicaremos esto partiendo de la gráfica de la Figura 2.3.

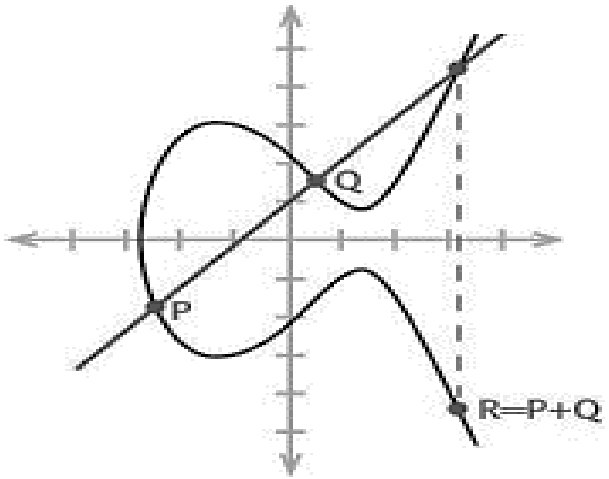


Figura 2.3. Generación de un punto en una curva elíptica.

Usamos como puntos de partida P y Q, dos puntos conocidos. Trazaremos una línea entre P y Q. Si la línea corta la curva en un tercer punto, lo reflejaremos a través del eje, dando lugar a un nuevo punto R. Esta operación se representa como R=P+Q. En caso de que la línea que pasa por P y Q no corte a la curva en ningún otro punto, diremos que corta la curva en un punto O en el infinito y representaremos esta operación como P+Q=O.

Partiendo de la suma, no es difícil encontrar un mecanismo que nos permita realizar multiplicaciones de tipo kP, siendo k un escalar. Por ejemplo, imaginemos que

queremos realizar la operación 13P, es decir, multiplicar 13 por un punto P. Bastaría con realizar la siguiente secuencia de doblado de puntos:

P, 2P=P+P, 4P=2P+2P, 8P=4P+4P, 13P=8P+4P+P

Este simple mecanismo para generar nuevos puntos dota a una curva elíptica de la posibilidad de realizar operaciones aritméticas sobre ella, base de los criptosistemas que usan curvas elípticas.

2.2.2 Funcionamiento e implementación

A continuación veremos una pequeña explicación matemática de cómo implementar un criptograma elíptico para la generación de claves:

En primer lugar se deben elegir un campo finito subyacente GF(q), una curva elíptica E definida sobre GF(q), y un punto P sobre E de orden primo n. El campo GF(q), la curva E, el punto P y el orden n forman los parámetros del sistema y son públicos. Para generar sus claves, cada principal en el sistema debe realizar el siguiente procedimiento:

1. Elegir un número entero aleatorio d tal que $1 \leq d < n$.
2. Calcular el punto Q = dP.
3. La clave pública del principal consiste en el punto Q.
4. La clave privada del principal es el entero d.

Si B desea enviar el mensaje M en forma cifrada a A debe realizar el procedimiento que se describe a continuación:

1. Obtener la clave pública Q de A.
2. Representar al mensaje M por un elemento del campo $m \in GF(q)$.
3. Seleccionar un entero aleatorio k tal que $1 \leq k < n$.
4. Calcular el punto (x1, y1) = kP.
5. Calcular el punto (x2, y2) = kQ. Si x2 = 0 entonces ir al paso 3.
6. Calcular c = m to (x
7. Transmitir el dato cifrado (x1, y1, c) a A.

Una vez que A recibe el mensaje cifrado (x1, y1, c) de B, para descifrarlo realiza los siguientes pasos:

1. Calcula el punto (x2, y2) = d(x1, y1), usando su clave privada d.
2. Recupera el dato m calculando $m = c \cdot x^{-1} 2$.

Supongamos que A desea firmar el mensaje M a enviar a B. Entonces A debe realizar el siguiente procedimiento:

1. Usar un algoritmo de hash para calcular el valor de hash e = H(M).
2. Seleccionar un entero aleatorio k tal que $1 \leq k < n$.
3. Calcular el punto (x1, y1) = kP y hacer r= x1 mod n.
4. Usar su clave privada para calcular $s = k^{-1} (e + rd) \bmod n$.
5. A envía a B el mensaje M y su firma (r, s).

Si B desea verificar la firma (r , s) de A para el mensaje M, debe realizar los siguientes pasos:

1. Obtener la clave pública Q de A.
2. Si (r mod n) = 0 entonces rechazar la firma.
3. Calcular el valor de hash e := H(M).
4. Calcular $s^{-1} \bmod n$.
5. Calcular $u := s^{-1}e \bmod n$ y $v := s^{-1}r \bmod n$.
6. Calcular al punto (x1, y1):= uP + vQ.
7. Aceptar la firma de A para el mensaje M si y sólo si (x1 mod n) = r .

Si r = 0 entonces la ecuación de la firma $s = k^{-1}(e + rd)$ no involucra la clave privada d, y de ahí la condición en el paso 4.

Los algoritmos criptográficos basados en curvas elípticas que se pueden usar actualmente son:

- **ElGamal**

Este protocolo se implementaría de la siguiente manera:

Sea E una curva elíptica sobre Z/p, sea P un punto de la curva de orden grande N (sería deseable que hPi = E), N|#E(p). Para cada usuario U, sea nU su clave privada, $1 < n_U < N$; (bastaría tomar $n_U < p + 1 - 2\sqrt{p}$). La clave pública de U será $P_U = n_U P$.

A quiere enviar el mensaje m cifrado al usuario B:

1. A escoge al azar un número $k \in Z/p$,
2. Calcula Pm el punto de la curva asociado al mensaje m,
3. Cifra Pm como $C = E_B(Pm) = Pm + k \cdot PB$,
4. Envía a B (C,kP).

B para descifrar el mensaje deberá hacer:

1. Pm = C – nB(kP),
2. encuentra el mensaje m asociado con el punto Pm.

- **ECDH (*Elliptic Curve Diffie-Hellman*)**

El intercambio de llaves EC Diffie-Hellman se realiza de la siguiente forma usando curvas elípticas:

Sea E una curva elíptica sobre Fp y P ∈ E punto públicamente conocido. Cada usuario U elige al azar un número secreto nU ∈ Fp y hace público el valor n_UP. Para compartir una clave secreta, A y B deben hacer:

$$A\ n_A P \rightarrow B$$

$$A\ n_B P \leftarrow B$$

La clave secreta será K = ($n_A \cdot n_B$)P que solo es conocida por A y B.

- **RSA**

En este esquema se representan los puntos de una curva elíptica de la forma $y^2 = x^3 + b$ sobre *Zn* como $E_n(b)$. Para generar la clave pública el usuario *B* escogerá dos números primos grandes (*p*,*q*) tales que $p = q = 2 \pmod 3$ y, como en el esquema clásico, calculará y publicará (*e*,*n*), donde $n = p \cdot q$ y mantendrá en secreto las claves privadas (*p*,*q*,'(*n*),*d*).

Cada vez que A quiere enviar un mensaje *m* a B deberá seguir los siguientes pasos:

1. El usuario A divide su mensaje *m* en dos partes *m = (m1,m2)* donde m1,m2 ∈ Zn.
2. El usuario A determina el valor b de la curva de forma que m ∈ En(b). Específicamente, calcula b = $m^2{}_2 - m^3{}_1 \pmod n$.
3. Cifra el punto m calculando c = E(m) = e·m sobre $E_n(b)$,
4. Envía el texto cifrado c = (c1,c2) a B.

El usuario B para descifrar el mensaje c deberá hacer:

1. A partir del mensaje cifrado c = (c1,c2) el usuario B puede determinar el valor de b puesto que este no cambia en el proceso de cifrado. Específicamente, calcula b = $c^2{}_2 - c^3{}_1 \pmod n$ y construye la curva y2 = x3 + b.
2. A partir de la clave privada calcula m = D(c) = d·c sobre En(0,b).

- **ECDSA (*Elliptic Curve Digital Signature Algorithm*)**

EC Digital Signature Algorithm se ha convertido en el estándar de firma digital con curvas elípticas y por tanto es el protocolo cuyo funcionamiento estudiaremos en este proyecto. A continuación lo veremos con más detalle.

2.2.3 ECDSA

ECDSA (*Elliptic Curve Digital Signature Algorithm*) es una modificación del algoritmo DSA que emplea operaciones sobre puntos de curvas elípticas en lugar de las exponenciaciones que usa DSA (problema del logaritmo discreto). Los procesos de generación y verificación de firma se basan en la configuración del criptosistema ElGamal.

Este algoritmo permite firmar documentos y verificar las firmas. La principal ventaja es que requiere números de tamaños menores para ofrecer la misma seguridad que DSA o RSA.

- **Proceso de firma y verificación mediante ECDSA**

  - Generación de claves

    1. Selecciona una curva elíptica E.
    2. Selecciona un punto P (que pertenezca a E) de orden n.
    3. Selecciona aleatoriamente un número d en el intervalo [1, n - 1].
    4. Calcula Q = dP.
    5. d será la llave privada.
    6. Q será la llave pública.

  - Proceso de firma

    1. Selecciona un número k de forma aleatoria.
    2. Calcule $kP = (x_1, y_1)$.
    3. Calcula $r = x_1 \bmod n$. Si $r = 0$ regresa al primer paso. (En este paso x_1 es tratado como un entero).
    4. Calcula $(k^{-1}) \bmod n$.
    5. Calcula $s = k^{-1}(H(m) + dr) \bmod n$. Si $s = 0$ regrese al primer paso. (H(m) es el hash del mensaje a firmar, calculado con SHA-1).

6. La firma del mensaje m son los números r y s.

- Proceso de verificación

1. Verifica que r y s estén dentro del rango [1,n - 1].
2. Calcula $w = s^{-1} \bmod n$.
3. Calcula $u_1 = H(m)w \bmod n$.
4. Calcula $u_2 = r \cdot w \bmod n$.
5. Calcula $u_1P + u_2Q = (x_0,y_0)$
6. Calcula $v = x_0 \bmod n$
7. La firma verifica si y solo si v = r

- **Ventajas del uso de EDSA frente a RSA**

Como se mencionó previamente, el algoritmo ECDSA se ha convertido en el estándar de firma digital con curvas elípticas compitiendo en uso mano a mano con el algoritmo de firma digital más usado en la actualidad, RSA, esto es debido a que presenta ciertas características que hacen que su uso aporte ciertas ventajas con respecto a RSA.

- Seguridad:

Para llegar a un grado aceptable de seguridad RSA debería usar claves de 1024 bits, mientras que para la ECDSA sería suficiente con 160. A medida que la clave crece, aumenta la distancia entre la seguridad de cada uno. Por ejemplo, ECDSA con 380 bits es mucho más seguro que RSA con 2000 bits (de hecho, para esta longitud de clave, ECDSA es comparable a RSA de 7600 bits).

- Eficiencia

Para comparar los niveles de eficiencia, deberemos tener en cuenta:

1) Costes computacionales, o sea la cantidad de computación requerida para cifrar y descifrar. RSA y ECDSA exigen un gran esfuerzo computacional, sin embargo, teniendo en cuenta estudios actuales las implementaciones de ECDSA son un orden de magnitud más rápido que el RSA

2) Tamaño de la clave, o sea, la cantidad de bits necesarios para guardar la pareja de claves y los otros parámetros del sistema. En la Tabla 2.1 vemos los tamaños de clave equivalentes entre RSA y ECC (ECDSA):

RSA	ECDSA
163	1024
233	2048
283	3072
409	7680
571	15360

Tabla 2.1. Equivalencias entre claves RSA/ECDSA [4]

Vemos como los tamaños de clave de ECDSA con respecto a RSA para un mismo nivel de seguridad es mucho menor en el primero.

3) Ancho de banda, o sea, la cantidad de bits que se deben transmitir para comunicar un mensaje cifrado o una firma digital. RSA y ECDSA requieren el mismo ancho de banda cuando se usan para cifrar o firmar mensajes largos.

Los dos tipos de criptosistemas requieren el mismo ancho de banda para firmar mensajes largos. De todos modos cuando los mensajes no son largos se ha de observar con más atención (y, de hecho, este tipo de mensajes son los que usualmente son utilizados en la criptografía de clave pública). Para poder hacer comparaciones, suponemos que queremos firmar un mensaje de 2000 bits. La Tabla 2.2 compara las longitudes de las firmas:

	Tamaño de la firma (bits)
RSA	1024
ECDSA	320

Tabla 2.2. Equivalencias entre claves RSA/ECDSA para un mensaje de 2000 bits

En resumen, el sistema ECDSA tiene una gran eficiencia y, en las implementaciones, esto significa rapidez, bajo consumo y reducción de la medida del código transmitido.

2.3 Plataforma Android

Android es una plataforma de código libre basada en el núcleo de Google orientada a servir de sistema operativo en dispositivos móviles con pantalla táctil, aunque en la actualidad puede ser encontrada también en televisores, tablets, relojes inteligente y coches. La mayor parte del código Android está liberado por Google bajo licencia Apache.

Fue desarrollado por Android Inc, empresa que posteriormente fue adquirida por Google en 2005 encargándose de su desarrollo desde entonces. Vio la luz en 2007 junto con la presentación del consorcio Open Handset (un consorcio de compañías de hardware, *software* y telecomunicaciones para avanzar en los estándares abiertos de los dispositivos móviles, entre las que están Texas Instruments, Broadcom Corporation, Nvidia, Qualcomm, Samsung Electronics, Sprint Nextel, Intel, LG, Marvell Technology Group, Motorola, T-Mobile, PacketVideo, ARM Holdings, Atheros Communications, Asustek, Garmin, Softbank, Sony Ericsson, Huawei, Toshiba, Vodafone y ZTE). Podemos decir que Android es el primer producto de este consorcio. [5][6]

En la actualidad es el sistema operativo para dispositivos móviles más usado muy por encima de sus actuales competidores iOS y Windows Phone.

En la Figura 2.4 podemos ver la arquitectura básica de Android.

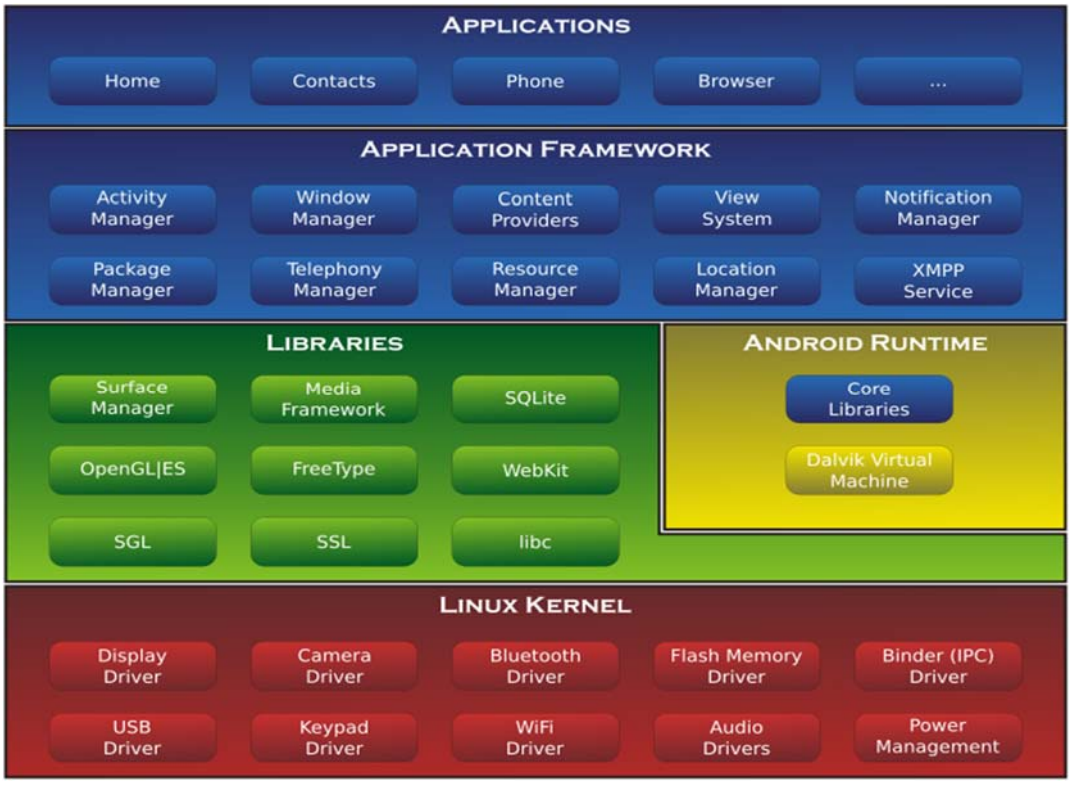


Figura 2.4. Arquitectura básica de Android [6]

La estructura del sistema operativo Android se compone de aplicaciones que se ejecutan en un *framework* Java sobre el núcleo de las bibliotecas de Java en una máquina virtual Dalvik con compilación en tiempo de ejecución. Las bibliotecas escritas en lenguaje C incluyen un administrador de interfaz gráfica (*surface manager*), un *framework* OpenCore, una base de datos relacional SQLite, una Interfaz de programación de API gráfica OpenGL ES 2.0 3D, un motor de renderizado WebKit, un motor gráfico SGL, SSL (Secure Socket Layer) y una biblioteca estándar de C Bionic.

Los desarrolladores tienen acceso completo a los APIs [7] del framework usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del framework). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario. Las aplicaciones se desarrollan habitualmente en el lenguaje Java con Android Software Development Kit (Android SDK) [8], pero están disponibles otras herramientas de desarrollo, incluyendo un Kit de Desarrollo Nativo para aplicaciones o extensiones en C o C++.

Desde su creación hasta la actualidad Android ha pasado por trece versiones nombradas desde la A hasta la M (en desarrollo actualmente)[1]. Suelen recibir nombres de diferentes postres o dulces. La última versión comercializada ha sido Android Lollipop (v5.0/5.1), siendo actualmente la más usada Android KitKat (v4.4)[2].

2.3.1 Seguridad en Android

Android propone un esquema de seguridad que protege a los usuarios, sin la necesidad de imponer un sistema centralizado. Se fundamenta en los siguientes tres pilares:

- Ejecución en caja de Arena:
  Android puede impedir que las aplicaciones tengan acceso directo al hardware o interfieran con recursos de otras aplicaciones. Android crea una cuenta de usuario Linux (user ID) nueva por cada paquete (.apk) instalado en el sistema. Este usuario es creado cuando se instala la aplicación y permanece constante durante toda su vida en el dispositivo. Cualquier dato almacenado por la aplicación será asignado a su usuario Linux, por lo que normalmente no tendrán acceso otras aplicaciones.

- Firma digital de las aplicaciones:
  Toda aplicación ha de ser firmada con un certificado digital que identifique a su autor. La firma digital también nos garantiza que el fichero de la aplicación no ha sido modificado. Si se desea modificar la aplicación está tendrá que ser

[1] https://developer.android.com/about/versions/marshmallow/index.html
[2] https://developer.android.com/about/versions/kitkat.html

firmada de nuevo, y esto solo podrá hacerlo el propietario de la clave privada. Es habitual que un certificado digital sea firmado a su vez por una autoridad de certificación, sin embargo en Android esto no es necesario.

- Esquemas de permisos:
  Para proteger ciertos recursos y características especiales, Android define un esquema de permisos. Toda aplicación que acceda a estos recursos está obligada a declarar su intención de usarlos. En caso de que una aplicación intente acceder a un recurso del que no ha solicitado permiso, se generará una excepción de permiso y la aplicación será interrumpida inmediatamente.

2.3.2 Gestión TLS en Android

Si miramos el gráfico de la arquitectura de Android (Figura 2.2) explicado anteriormente comprobamos que una de las librerías que usa Android es SSL. La capa SSL, ahora conocida técnicamente como *Transport Layer Security* (TLS), es un bloque común de construcción de las comunicaciones cifradas entre clientes y servidores que puede ser implementada para el uso de cualquier aplicación. De esto se puede deducir, que éste es el protocolo usado para realizar las conexiones seguras entre Android y cualquier servidor externo. [9]

En un escenario típico de uso de TLS, un servidor está configurado con un certificado que contiene una clave pública, así como una clave privada correspondiente. Como parte del *handshake* entre un cliente TLS y el servidor, el servidor prueba que tiene la clave privada mediante la firma de su certificado con la clave pública.

Sin embargo, cualquier persona puede generar su propio certificado y su clave privada, por lo que un sencillo apretón de manos no prueba nada sobre el servidor más que el servidor conoce la clave privada que coincide con la clave pública del certificado. Una manera de resolver este problema es hacer que el cliente tenga un conjunto de uno o más certificados en los que confía. Si el certificado no está en el conjunto, el servidor no es de fiar.

Este enfoque presenta varias desventajas. Los servidores deben ser capaces de actualizar a claves más fuertes con el tiempo ("rotación de clave"), que sustituye a la clave pública en el certificado con una nueva. Por desgracia, ahora la aplicación cliente tiene que ser actualizada debido al cambio de configuración del servidor. Esto es especialmente problemático si el servidor no está bajo el control del desarrollador de la aplicación, por ejemplo, si se trata de un servicio web de terceros. Este enfoque también tiene problemas si la aplicación tiene que hablar con servidores arbitrarios, como un navegador web o aplicación de correo electrónico.

Para hacer frente a estos inconvenientes, los servidores suelen ser configurados con certificados de emisores conocidos llamadas autoridades de certificación (CA). A partir de Android 4.2 (también llamada Android Jelly Bean), Android contiene más de 100 entidades emisoras de certificados que se actualizan en cada versión. Al igual que un servidor, una CA tiene un certificado y una clave privada. Al expedir un certificado para un servidor, la CA firma el certificado de servidor utilizando su clave privada. El cliente puede verificar que el servidor tiene un certificado emitido por una CA conocida por la plataforma.

Sin embargo, el uso de las CA introduce otros problemas. Debido a que la CA emite certificados para muchos servidores, todavía se necesita de alguna manera asegurarse de que está hablando con el servidor que desea. Para hacer frente a esto, el certificado emitido por la CA identifica el servidor, ya sea con un nombre específico, como *gmail.com* o un conjunto como *google.com*.

- **Implementación de una conexión TLS en Android**

Asumiendo que el servidor Web con el que nos queremos conectar tiene un certificado emitido por una CA conocida, podemos realizar una petición segura de la siguiente manera**:**

```
URL url =  new URL ( "https://google.com" );
URLConnection urlConnection = url . openConnection ();
InputStream  in  = urlConnection . getInputStream ();
copyInputStreamToOutputStream ( in ,  System . out );
```

Comprobamos que es la API de Android la encargada de realizar la verificación de los certificados y nombres de hosts.

Si quisiéramos realizar esto con nuestros propios servidores y certificados y no los tenemos firmados por ninguna de las CA que Android usa, tendremos que cargar y gestionar nosotros mismos los certificados. Aquí podemos diferenciar varios casos:

1) Autoridad de certificación desconocida

Esto ocurre cuando se tiene una CA que no sea de confianza para el sistema. Podría ser porque se tiene un certificado de una CA nueva que aún no sea de confianza para Android o la aplicación se está ejecutando en una versión anterior sin la CA. Lo más habitual es que la CA sea de una entidad privada para su propio uso. Afortunadamente, se puede enseñar a HttpsURLConnection a confiar en un conjunto específico de entidades emisoras. Se tiene una CA específica desde un InputStream que utiliza para crear un almacén de claves, que luego se usa para crear e inicializar un TrustManager. Un TrustManager es lo que utiliza el sistema para validar los

certificados del servidor. Con el nuevo TrustManager, se inicializa un nuevo SSLContext que proporciona una SSLSocketFactory, que sustituye el valor predeterminado de SSLSocketFactory. De esta manera la conexión utilizará las CA que queremos para la validación de certificados. A continuación podemos ver el ejemplo por defecto que Google nos da para implementarlo:

```
// Load CAs from an InputStream
// (could be from a resource or ByteArrayInputStream or ...)
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// From https://www.washington.edu/itconnect/security/ca/load-der.crt
InputStream caInput = new BufferedInputStream(new FileInputStream("load-der.crt"));
Certificate ca;
try {
    ca = cf.generateCertificate(caInput);
    System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
} finally {
    caInput.close();
}
// Create a KeyStore containing our trusted CAs
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);
// Create a TrustManager that trusts the CAs in our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);
// Create an SSLContext that uses our TrustManager
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);
// Tell the URLConnection to use a SocketFactory from our SSLContext
URL url = new URL("https://certs.cac.washington.edu/CAtest/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setSSLSocketFactory(context.getSocketFactory());
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

2) Certificados autofirmados.

En este caso el servidor se está comportando como su propia CA. Esto es similar a una autoridad de certificación desconocida, por lo que se utiliza el mismo enfoque del punto anterior.

3) Falta CA intermedia

Se produce debido a una CA intermedia que falta. La mayoría de las entidades emisoras públicas no firman los certificados del servidor directamente. En su lugar, utilizan su certificado de CA principal, denominada la CA raíz, para firmar las CA intermedias, por lo que la CA raíz se puede almacenar sin conexión para reducir el riesgo. Sin embargo, los sistemas operativos como Android normalmente confían sólo en la CA raíz, lo que deja poca confianza entre el certificado firmado por la CA-intermedio y el certificado de servidor, que conoce la CA raíz. Para solucionar esto, el servidor envía al cliente, durante el enlace SSL, una cadena de certificados del servidor

de CA de cualquier intermediario necesario para llegar a una CA raíz de confianza. Hay dos enfoques para resolver este problema:

- Configurar el servidor para incluir la CA intermedia en la cadena de servidor. La mayoría de las CA proporcionan documentación sobre cómo hacer esto para todos los servidores web comunes. Este es el único enfoque si se necesita el sitio para trabajar con los navegadores por defecto de Android, al menos, a partir de Android 4.2.
- O bien, tratar el CA intermedia como cualquier otra CA desconocida, y crear un TrustManager, como se hizo en los dos puntos anteriores.

4) Problemas con la verificación del nombre de host

Esto ocurre porque el servidor está configurado con un certificado que no tiene campos de nombre que coinciden con el servidor al que está tratando de alcanzar. Es posible tener un certificado que se pueda utilizar con muchos servidores diferentes. El error se produce sólo cuando el nombre del servidor que se está conectando no aparece en el certificado como aceptable. Esto también puede suceder por el uso de *hosts virtuales*. Una solución es la creación de una máquina virtual alternativa en un puerto único. La alternativa más drástica es reemplazar HostnameVerifier con uno que no utiliza el nombre de host de la máquina virtual, pero que devuelve el servidor de forma predeterminada. Lo podemos implementar de la siguiente manera:

```
// Create an HostnameVerifier that hardwires the expected hostname.
// Note that is different than the URL's hostname:
// example.com versus example.org
HostnameVerifier hostnameVerifier = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession session) {
        HostnameVerifier hv =
            HttpsURLConnection.getDefaultHostnameVerifier();
        return hv.verify("example.com", session);
    } };
// Tell the URLConnection to use our HostnameVerifier
URL url = new URL("https://example.org/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setHostnameVerifier(hostnameVerifier);
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Con el fin de mitigar el riesgo a una verificación de host incorrecta, Android posee una lista negra con ciertos certificados o incluso CAs. Si bien esta lista fue construida en el sistema operativo, a partir de Android 4.2 esta lista se puede actualizar de forma remota para hacer frente a problemas futuros.

2.3.3 Navegadores Web para Android

En Android cualquier aplicación puede conectarse a Internet siempre y cuando cumpla con las condiciones que el API de Android pone para realizar conexiones. Las principales aplicaciones y más usadas para conectarse a Internet son los navegadores web. Éstos, a semejanza de sus hermanos mayores, los usados en PCs (Chrome, Firefox, Explorer, etc), ofrecen una suite que permite conectarse a través de la red a servidores comerciales (y no comerciales).

Además de permitir conectarse a cualquier servidor externo deben implementar una serie de protocolos que permitan conexiones seguras con estos servidores pues la mayoría de las veces estas conexiones están destinadas al intercambio de datos confidenciales que sería peligroso si fueran interceptados por un tercero. Es por ello que estos navegadores ofrecen diferentes entornos seguros de conexión permitiendo el uso de diferentes tipos de algoritmos y protocolos para realizar estas conexiones. Siempre, por supuesto, que estos algoritmos y protocolos estén soportados por Android y por el servidor al que quieren conectarse.

En capítulos posteriores probaremos el soporte de estos navegadores a las *suites* de cifradores que la mayoría de los servidores soportan. Este soporte estará limitado primero por los cifradores que soporta Android y en segundo lugar y definitivo por los que soporta el propio navegador.

Los navegadores comerciales más usados actualmente en Android [10] son:

- **Chrome**[3]. El navegador web para dispositivos móviles por defecto de Google. Es el más descargado con más de 500 millones de descargas, aunque gran parte de ese mérito es que es el navegador por defecto de Android desde la versión 4.4 (Android KitKat). Destaca en la sincronización de marcadores y pestañas entre dispositivos, su traductor de páginas webs, su modo para ahorrar datos y su modo incógnito para navegar de forma privada.

- **Dolphin**[4]. El navegador Dolphin fue uno de los primeros en llegar a Android y de los más completos. Incluye sincronización entre dispositivos, extensiones, tienda de aplicaciones web, temas, soporte a Adobe Flash, modo privado, modo nocturno y gestos para acceder a páginas webs favoritas dibujando una letra.

  Opera[5]. Basado en Chrome, es un navegador algo básico que incluye navegación privada, una sección para descubrir noticias y su famoso modo todo terreno basado en Opera Turbo para reducir los datos de navegación.

[3] https://www.google.es/chrome/browser/mobile/
[4] http://dolphin.com/es/features-7/
[5] http://www.opera.com/es/mobile/operabrowser/android

- **Firefox**[6]. La fundación Mozilla nos ofrece con Firefox para Android un completo navegador web que cuenta con paneles de inicio personalizables, sincronización entre dispositivos, modo incógnito, sesión para invitados, guardar páginas como PDF, complementos y Firefox Marketplace para instalar aplicaciones y juegos web.

- **UC Browser**[7]. UC Browser destaca por su navegación rápida, navegación de incógnito, complementos, temas, modo noche, control de vídeo por gestos y en sus sus notificaciones de Facebook en tiempo real con el navegador cerrado.

- **CM Browser**[8]. CM Browser nos ofrece un navegador seguro, veloz y ligero. Es el que menos tamaño ocupa. Su descarga es de menos de 2 MB y una vez instalado ocupa menos de 10 MB. Cuenta con pre-carga para agilizar la navegación, modo incógnito y protección de descargas maliciosas.

- **Maxthon**[9]. Este navegador es el ganador del premio al mejor navegador durante 3 años consecutivos. Cuenta con sincronización de pestañas y marcadores entre dispositivos y en su versión para PC, modo de navegación privada, modo lectura y permite descargar archivos directamente en la nube.

[6] https://www.mozilla.org/es-ES/firefox/android/
[7] http://es.ucweb.com/ucbrowser/
[8] http://www.cmcm.com/en-us/cm-browser/
[9] http://es.maxthon.com/

2.4 Servidores Web

Un servidor web es un programa informático que procesa una aplicación del lado del servidor, realizando conexiones bidireccionales y/o unidireccionales y síncronas o asíncronas con el cliente y generando o cediendo una respuesta en cualquier lenguaje o aplicación del lado del cliente.

En la Figura 2.5 podemos ver un ejemplo de servidor HTTP.

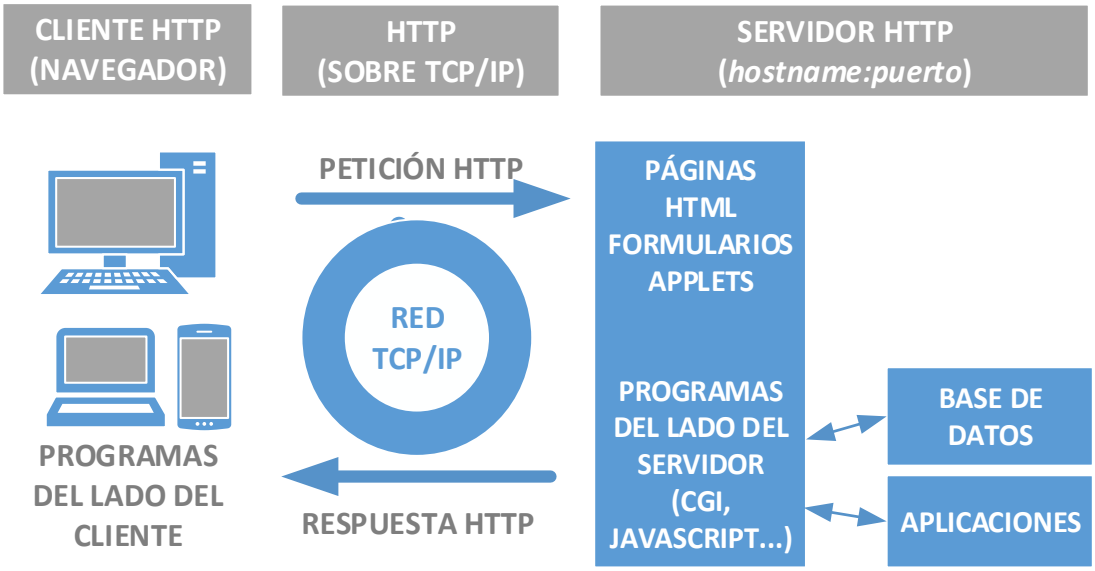


Figura 2.5. Arquitectura de un servidor HTTP

- **Partes de un servidor web**

Las principales partes de cualquier servidor web son:

- **Servidor HTTP**: Procesan mensajes HTTP (protocolo de comunicación) de clientes y devuelven mensajes con la información solicitada (Estados, datos, códigos de error….). Todas las operaciones (peticiones y respuestas) pueden adjuntar recursos web (Documentos HTML, ficheros multimedia, aplicaciones) Los servidores HTTP más utilizados son: Microsoft Information Server, Nginx y Apache [11]. Éste último es el más usado con diferencia con respecto al resto. En el siguiente apartado veremos algunas de las características a las que se debe su gran implantación.

- **Lenguaje de *Script***: Un lenguaje de *script* es el lenguaje de programación con el que se escriben los *scripts* (guión o archivo de órdenes escritos normalmente en texto plano). Los *scripts* son interpretados por un intérprete (o traductor) que se encarga de traducir las órdenes escritas en ellos, para que puedan ser ejecutadas. El resultado de ejecutar, el servidor lo devuelve al navegador como HTML.

- **Base de datos**: Un servidor necesita una herramienta donde guardar los datos que pueda necesitar para responder las peticiones del cliente. El uso de bases de datos presenta muchas ventajas como la posibilidad de guardar la información manteniendo relaciones de unos datos con otros, la búsqueda relativamente sencilla de cualquier dato y el control de qué servicios acceden a la información.

- **Funcionamiento básico de un servidor web**

El servidor, se encarga de recibir las peticiones y gestionarlas, buscar en el repositorio de páginas (almacén donde están guardadas) la página y con un intérprete obtiene las órdenes necesarias para crear la respuesta que enviará al navegador. Para crear esa respuesta también necesitará consultar una base de datos para obtener los datos necesarios para crear la respuesta y enviarla. Finalmente el navegador interpreta el código HTML y presenta la web.

2.4.1 Servidor Apache

El servidor HTTP Apache es un servidor web HTTP de código abierto, para plataformas Unix (BSD, GNU/Linux, etc.), Microsoft Windows, Macintosh y otras, que implementa el protocolo HTTP/1.12 y la noción de sitio virtual. Se desarrolla dentro del proyecto HTTP Server (httpd) de la *Apache Software Foundation*[10].

La arquitectura del servidor Apache es muy modular. El servidor consta de un núcleo y diversos módulos que aportan mucha de la funcionalidad básica de un servidor web. Algunos de estos módulos son:

- `mod_ssl`: Comunicaciones Seguras vía TLS.
- `mod_rewrite`: Reescritura de direcciones. Utilizado para transformar páginas dinámicas como php en páginas estáticas html.

[10] http://httpd.apache.org/

- `mod_dav`: Soporte del protocolo WebDAV (RFC 2518).
- `mod_deflate`: Compresión transparente con el algoritmo deflate del contenido enviado al cliente.
- `mod_auth_ldap`: Permite autentificar usuarios contra un servidor LDAP.
- `mod_proxy_ajp`: Conector para enlazar con el servidor Jakarta Tomcat de páginas dinámicas en Java.
- `mod_cfml`: Conector CFML usado por Railo.

Esta funcionalidad básica puede ser extendida mediante el uso de módulos externos. Algunos ejemplos son:

- `mod_cband`: Control de tráfico y limitador de ancho de banda.
- `mod_perl`: Páginas dinámicas en Perl.
- `mod_php`: Páginas dinámicas en PHP.
- `mod_python`: Páginas dinámicas en Python.
- `mod_rexx:` Páginas dinámicas en REXX y Object REXX.
- `mod_ruby`: Páginas dinámicas en Ruby.
- `mod_aspdotnet`: Páginas dinámicas en .NET de Microsoft (Módulo retirado).
- `mod_mono`: Páginas dinámicas en Mono.
- `mod_security`: Filtrado a nivel de aplicación, para seguridad.

La mayor parte de la configuración se realiza en el fichero `apache2.conf` (Debian, Ubuntu) o `httpd.conf` (Otros). Se realiza mediante el uso de directivas que modifican parámetros. Esto se puede realizar sobre los propios archivos de configuración o mediante comandos en la consola del sistema operativo sobre el que esté funcionando Apache.

Apache es usado principalmente para el envío de páginas web estáticas y dinámicas en la red. Habitualmente es redistribuido como parte de varios paquetes propietarios de *software*, como por ejemplo en Android. También es usado para muchas otras tareas donde el contenido necesita ser puesto a disposición en una forma segura y confiable.

2.4.2 Conexiones seguras en Apache

Para realizar conexiones seguras Apache utiliza el protocolo TLS [12]. Para iniciar una conexión segura mediante TLS primero se realiza el proceso de handshake. Durante dicho proceso se negocían las características de la conexión segura. En la Figura 2.6 se observan los

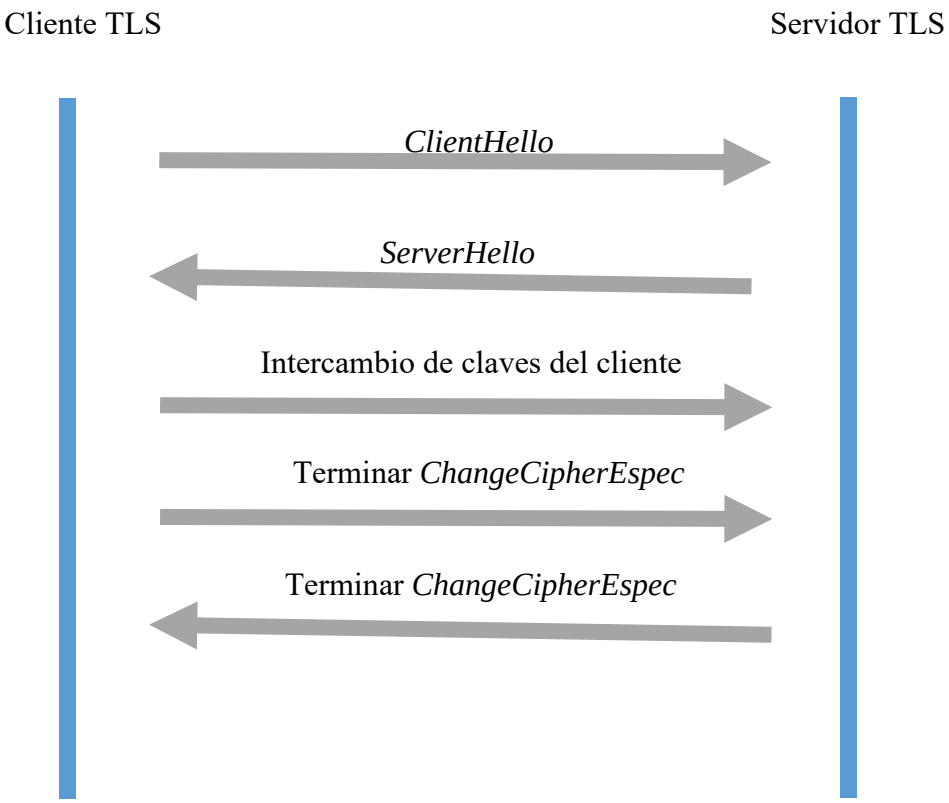


Figura 2.6. Handshake TLS

El protocolo TLS intercambia registros (Figura 2.6) que, opcionalmente, pueden ser comprimidos, cifrados y empaquetados con un código de autenticación del mensaje (MAC). Cada registro tiene un campo de *content_type* que especifica el protocolo de nivel superior que se está usando.

Cuando se inicia la conexión, el nivel de registro encapsula otro protocolo, el protocolo *handshake* que tiene el *content_type*. El cliente envía y recibe varias estructuras *handshake*:

- Envía un mensaje *ClientHello* especificando una lista de conjunto de cifrados, métodos de compresión y la versión del protocolo TLS/SSL más alta permitida. Éste también envía bytes aleatorios que serán usados más tarde (llamados *Challenge de Cliente* o *Reto*). Además puede incluir el identificador de la sesión.

- Después, recibe un registro *ServerHello*, en el que el servidor elige los parámetros de conexión a partir de las opciones ofertadas con anterioridad por el cliente.
- Cuando los parámetros de la conexión son conocidos, cliente y servidor intercambian certificados (dependiendo de las claves públicas de cifrado seleccionadas). Estos certificados son actualmente X.509, pero hay también un borrador especificando el uso de certificados basados en OpenPGP.
- Cliente y servidor negocian una clave secreta (simétrica) común llamada *master secret*, posiblemente usando el resultado de un intercambio Diffie-Hellman, o simplemente cifrando una clave secreta con una clave pública que es descifrada con la clave privada de cada uno. Todos los datos de claves restantes son derivados a partir de este *master secret* (y los valores aleatorios generados en el cliente y el servidor), que son pasados a través una *función pseudoaleatoria* cuidadosamente elegida.

- **Seguridad en TLS**

TLS posee una gran variedad de medidas de seguridad:

- Numerando todos los registros y usando el número de secuencia en el MAC.
- Usando un resumen de mensaje mejorado con una clave (de forma que solo con dicha clave se pueda comprobar el MAC).
- Protección contra varios ataques conocidos (incluyendo ataques *man-in-the-middle*), como los que implican un degradado del protocolo a versiones previas (por tanto, menos seguras), o conjuntos de cifrados más débiles.
- El mensaje que finaliza el protocolo *handshake* (*Finished*) envía un *hash* de todos los datos intercambiados y vistos por ambas partes.
- La función pseudo aleatoria divide los datos de entrada en 2 mitades y las procesa con algoritmos hash diferentes (MD5 y SHA), después realiza sobre ellos una operación XOR. De esta forma se protege a sí mismo de la eventualidad de que alguno de estos algoritmos se revelen vulnerables en el futuro.

- **Intercambio de clave**

Antes de que un cliente y un servidor puedan empezar a intercambiar información protegida por TLS, deben intercambiar de forma segura o acordar una clave de cifrado y una clave para usar cuando se cifren los datos. Entre los métodos utilizados para el intercambio/acuerdo de claves son: las claves públicas y privadas generadas con RSA (denotado TLS_RSA en el protocolo de handshake TLS), Diffie-Hellman (llamado TLS_DH), Diffie-Hellman efímero (denotado TLS_DHE), Diffie-

Hellman de Curva Elíptica (denotado TLS_ECDH), Diffie-Hellman de Curva Elíptica efímero (TLS_ECDHE), Diffie-Hellman anónimo (TLS_DH_anon), y PSK (TLS_PSK).

El método de acuerdo de claves TLS_DH_anon no autentica el servidor o el usuario y por lo tanto rara vez se utiliza puesto que es vulnerable a un ataque de suplantación de identidad. Sólo TLS_DHE y TLS_ECDHE proporcionan secreto-perfecto-hacia-adelante.

Los certificados de clave pública que se utilizan durante el intercambio/acuerdo también varían en el tamaño de las claves de cifrado públicas/privadas utilizadas durante el intercambio y, por tanto, en la solidez de la seguridad que proveen.

2.4.3 OpenSSL

Podemos definir OpenSSL como un robusto paquete de herramientas de administración y bibliotecas que suministran funciones criptográficas a otros paquetes como OpenSSH y navegadores web (para acceso seguro a sitios HTTPS)[11].

Se trata de un proyecto de software libre basado en SSLeay, desarrollado por Eric Young y Tim Hudson. Su última versión es la 1.0.2c.; aunque actualmente la más usada es la 1.0.1g, la cual ha sido actualizada con diferentes parches para solucionar ciertos problemas de seguridad.

Las herramientas que posee OpenSSL ayudan al sistema a implementar TLS. OpenSSL también permite crear certificados digitales que pueden aplicarse a un servidor. Es por ello que OpenSSL es usado para la creación de certificados de diferentes tipos para Apache.

Usado en Apache permite la creación de claves privadas con el tamaño de clave que queramos, de CSR (*Certificate Signing Request*) y a partir de estos dos, el certificado SSL, usando el protocolo de clave que queramos, como por ejemplo RSA, DSA o ECDSA.

2.4.4 Certificados X.509

Los certificados creados por OpenSSL se crean mayoritariamente usando el estándar de certificados X.509 [13]. Este estándar establece la estructura de los certificados de clave pública independientemente del algoritmo o protocolo de seguridad establezcan estos certificados.

[11] https://www.openssl.org/

La estructura de un certificado digital X.509 v3 es la siguiente:

Certificado

Versión

Número de serie

ID del algoritmo

Emisor

Validez

- No antes de

-No después de

Sujeto

Información de clave pública del sujeto

Algoritmo de clave pública

Clave pública del sujeto

Identificador único de emisor (opcional)

Identificador único de sujeto (opcional)

Extensiones (opcional)

Algoritmo usado para firmar el certificado

Firma digital del certificado

Su sintaxis se define empleando el lenguaje ASN.1 (*Abstract Syntax Notation One*). Siguiendo la notación de ASN.1, un certificado contiene diversos campos, agrupados en tres grandes grupos:

- El primer campo es el *subject* (sujeto), que contiene los datos que identifican al sujeto titular. Estos datos están expresados en notación DN (Distinguished Name), donde un DN se compone a su vez de diversos campos, siendo los más frecuentes los siguientes; CN (*Common Name*), OU (*Organizational Unit*), O (*Organization*) y C (*Country*). Además del nombre del sujeto titular (*subject*), el certificado, también contiene datos asociados al propio certificado digital, como la versión del certificado, su identificador (*serialNumber*), la CA firmante (issuer), el tiempo de validez (*validity*), etc. La versión X.509.v3 también permite utilizar campos opcionales (nombres alternativos, usos permitidos para la clave, ubicación de la CRL y de la CA,etc.).

- En segundo lugar, el certificado contiene la clave pública, que expresada en notación ASN.1, consta de dos campos, en primer lugar, el que muestra el algoritmo utilizado para crear la clave (ej. RSA), y en segundo lugar, la propia clave pública.

- Por último, la CA, ha añadido la secuencia de campos que identifican la firma de los campos previos. Esta secuencia contiene tres atributos, el algoritmo de firma utilizado, el hash de la firma, y la propia firma digital.

Las extensiones de archivo más importantes de certificados X.509 son:

- .CER: Certificado codificado en CER, algunas veces es una secuencia de certificados
- .DER: Certificado codificado en DER
- .PEM: Certificado codificado en Base64, encerrado entre "-----BEGIN CERTIFICATE-----" y "-----END CERTIFICATE-----"

Capítulo 3

METODOLOGÍA

En este capítulo detallaremos las configuraciones realizadas para implementar el estudio de tiempo y energía de las diferentes conexiones que queremos probar. Para ello se plantean una metodología según los siguientes objetivos:

1. Probar soporte de los navegadores comerciales móviles respecto a la suite de cifradores (*ciphersuite*) que soporta un servidor Web Apache con soporte de OpenSSL.

2. Probar la eficiencia de tiempo en conexiones HTTPS de los navegadores comerciales respecto a la suite de cifradores que soportan mediante la descarga de archivos de diferentes tamaños desde el servidor Web Apache con soporte OpenSSL.

3. Usando un cliente Android separar el proceso de *handshake* del de intercambio de datos para hacer medidas de rendimiento, a través del consumo de tiempo y de energía. En este punto se intenta encontrar:

- La diferencia en coste temporal/energético entre el uso de certificados RSA respecto a certificados con ECC.

- Las diferencias en coste temporal/energético utilizando distintos tamaños de claves para ambos tipos de certificados.

4. Por último, analizar el soporte de certificados con ECC en servidores comerciales bien conocidos y que utilizan HTTPS.

3.1 Entorno de medida

Para realizar las pruebas hemos diseñado el entorno de medida de la Figura 3.1.

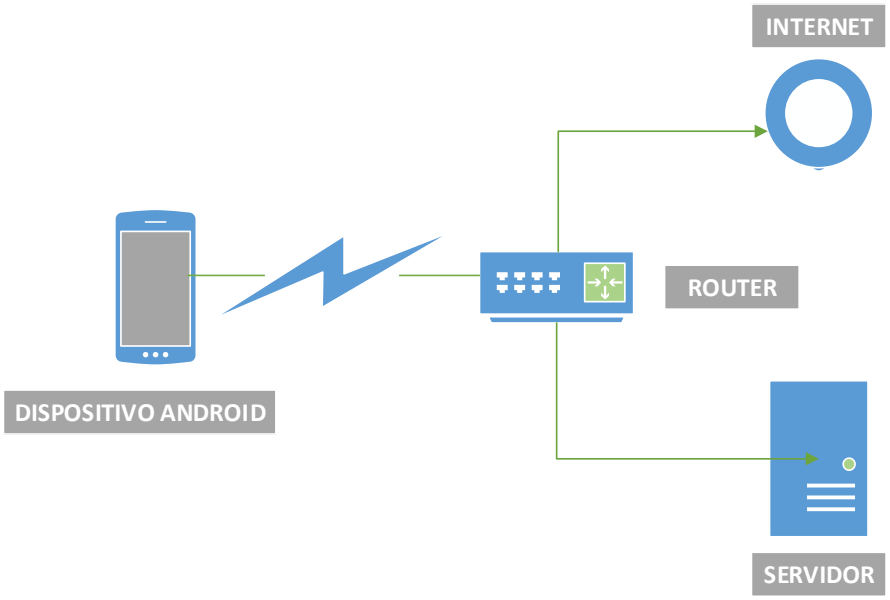


Figura 3.1. Esquema entorno de medida

A continuación veremos en detalle cada uno de los elementos usados y programados.

3.1.1 Dispositivos

Para la realización de las pruebas hemos usado un PC, un dispositivo Android y un router en un escenario aislado.

El PC ha sido empleado para el desarrollo y para las comunicaciones con el dispositivo Android. Presenta las siguientes características:

- Procesador Intel Core™ i3 CPU M380 @2.53GHZ.
- RAM 4,00 GB (3,86 utilizable).
- Sistema operativo Linux Ubuntu 15.04 de 64bits[12].
- Interfaz inalámbrica Atheros con driver ath5k para IEEE a/b/g

Como dispositivo Android hemos empleado un smartphone de la empresa bq, exactamente el modelo Aquaris 5[13] (Figura 3.2) con Android 4.4.2 que opera sobre la

[12] https://wiki.ubuntu.com/VividVervet/ReleaseNotes
[13] http://www.bq.com/es/support/aquaris-5

versión 3.4.67 del Kernel de Linux. En la Tabla 3.1 se detallan los componentes del dispositivo.

Figura 3.2. bq Aquaris 5 © bq.com

Pantalla	IPS de 5 pulgadas, multitáctil 5 puntos, 178º de visión
Resolución	qHD 540×960 píxeles, 220ppi
Procesador	Mediatek MT6589 Quad Core 1.2 GHz (ARMv7)
Procesador gráfico	PowerVR SGX 544MP
RAM	1 GB
Memoria	16 GB + microSD
Versión	Android 4.4.2 Kit Kat
Conectividad	Wireless 802.11b/g/n, Bluetooth, Dual SIM 3G+ (HSPA 24: 42 Mbps)
Puertos de expansión	MicroUSB, MicroUSB OTG, MicroSD
Cámaras	Frontal: VGA / Trasera: 8Mpx
Batería	Batería Li-ion 2200 mAh

Tabla 3.1. Especificaciones bq Aquaris 5

Para las conexiones entre el dispositivo Android y los servidores usamos un router inalámbrico que permite conectar dicho dispositivo con Internet y/o la creación de una red aislada entre el dispositivo y un servidor a través de él. Para ello se ha usado un router Amper ASL-26555[14] (Figura 3.3) conectado al PC mediante un cable RJ45 y a la red (cuando no se configura la red aislada) mediante una cable RJ11. Dicho router presenta las siguientes características:

- Estándar ADSL 2+ (G.992.5)
- WiFi IEEE 802.11n (compatible con 802.11b/g)
- Botón WPS
- 4 puertos Ethernet 10/100 Mbps
- Puerto USB
- Dimensiones 210 x 150 x 35 milímetros y peso 350 gramos

Figura 3.3. Router ASL-26555 © movistar.es

3.1.2 Herramientas software

Además de los dispositivos físicos descritos anteriormente, para la realización de las pruebas ha sido necesario el uso de diferentes herramientas de software. A continuación tenemos una descripción de cada una de ellas.

- **Servidor Apache** (v 2.4.10)

El servidor Apache[15] se encuentra en la mayoría de los repositorios de las distribuciones de Linux. En nuestro caso bastó con poner en consola:

```
$>> sudo apt-get install apache2
```

Tras este comando se instala la última versión de Apache, la versión 2.4.10, disponible para la versión de la distribución Linux que estamos usando, Ubuntu 15.04

[14] http://www.movistar.es/particulares/atencion-cliente/internet/adsl/equipamiento-adsl/routers/home-station-amper-asl26555/
[15] http://httpd.apache.org/docs/2.4/

de 64 bits. Al utilizar Ubuntu que está basado en Debian, la distribución de los ficheros de apache cambia con respecto a otras distribuciones [14]. En este caso los archivos de configuración los encontramos en *etc/apache2/* y los archivos de log en */var/log/apache2/*.

Para la realización de las diferentes medidas nos interesan los siguientes archivos de configuración:

- *default-ssl.conf*
- *000-default.conf*

Localizados ambos en la ruta:

- *etc/apache2/sites-avalables*

En estos archivos hemos ido variando diferentes parámetros de las conexiones, según el elemento que estemos evaluando en ese momento. En el primero se encuentra la configuración de las conexiones TLS y en el segundo la configuración general de las conexiones del servidor.

- **OpenSSL**

OpenSSL [20] es una herramienta que permite la creación de claves privadas con el tamaño de clave que queramos, de CSR (*Certificate Signing Request*) y a partir de estos dos, el certificado TLS, usando el protocolo de clave que queramos, en nuestro caso RSA y ECDSA. OpenSSL viene instalado por defecto en la versión de la distribución Linux que estamos usando. En nuestro caso se trata de la versión 1.0.1f[16].

En capítulos posteriores veremos cómo crear estos certificados mediante OpenSSL.

- **PowerTutor [15]**

Aplicación Android realizada por la universidad de Michigan. La podemos descargar de la web del desarrollador [17] o de la tienda de aplicaciones de Android (Play Store)[18].

Se encarga de realizar diferentes medidas de consumo de batería sobre el uso que hacen las aplicaciones que en ese momento se encuentran activas en Android. Si observamos la Figura 3.4 vemos el aspecto que presenta la interfaz de la aplicación.

[16] https://launchpad.net/ubuntu/+source/openssl/1.0.1f-1ubuntu2.15
[17] http://ziyang.eecs.umich.edu/projects/powertutor/
[18] https://play.google.com/store/apps/details?id=edu.umich.PowerTutor&hl=es

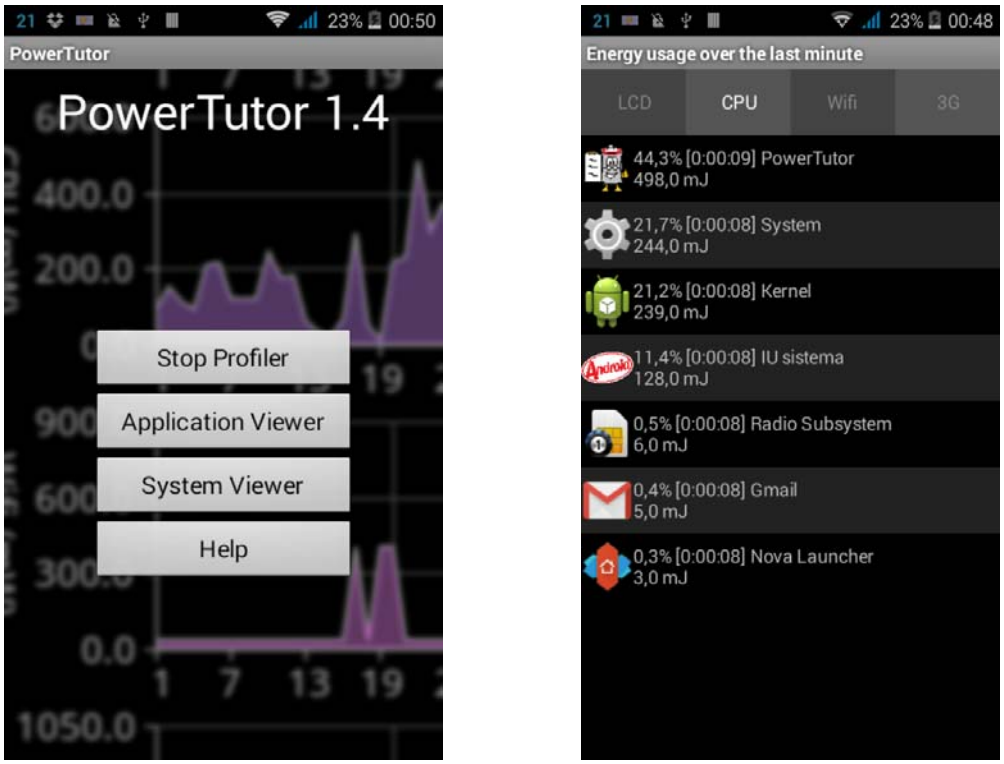


Figura 3.4. Interfaz PowerTutor

Mediante esta aplicación podemos estimar la energía consumida con cada conexión que realicemos desde el cliente Android pues mide la energía consumida por cada aplicación en el intervalo de tiempo que establezcamos dentro de las opciones disponibles. Gracias a ella realizaremos las mediciones de energía en las conexiones TLS mediante certificados RSA y ECDSA.

- **Android Studio**

Para realizar la producción de este programa se ha usado la suite de programación aportada por Google, Android Studio[19], la cual aporta las herramientas necesarias para realizar desarrollos en esta plataforma. Se ha usado la última versión disponible, concretamente la versión 1.4.1.

Además de la suite de programación, junto a ella se ha instalado el SDK de Android [8] que contiene todas las librerías necesarias para realizar una aplicación para esta plataforma.

Toda la información sobre esto, además del API de Android la encontramos en la Web Offcial de desarroladores de Android[20].

[19] https://developer.android.com/sdk/index.html
[20] https://developer.android.com

- **Navegadores Android**

Hemos instalado en nuestro dispositivo los cuatro navegadores Android más usados [17] para la realización de las pruebas que los necesitaban. Estos son: Chrome (navegador por defecto de Android), Firefox, Dolphin y Opera.

Han sido descargados de la tienda de aplicaciones oficial de Google (Google Play) e instalados en nuestro dispositivo móvil Android. Como única configuración adicional de cada uno de ellos además de la establecida por defecto ha sido permitir el uso de la apertura de pantallas adicionales, ya que las necesitaremos para algunas de las pruebas.

3.1.3 Software programado

Para realizar las conexiones desde Android al servidor necesitamos un navegador que realice conexiones TLS con el servidor Apache o con servidores comerciales y que sea capaz de establecer algunos de los parámetros de las conexiones. También necesitamos que sea capaz de medir ciertos valores de las conexiones y guarde esta información.

Para ello se ha desarrollado un navegador en Android. Tal y como comentamos anteriormente, para ello se ha usado la suite de programación Android Studio [16].

Una vez instalado todo lo necesario para comenzar a realizar la aplicación, establecimos qué necesita el navegador que queremos implementar:

- Capacidad de conexión TLS con el servidor Apache que tenemos funcionando.

- Gestión de certificados RSA y ECDSA.

- Capacidad de conexiones TLS con servidores comerciales.

- Capacidad de establecer diversos valores de las conexiones (cifradores a usar, certificado, etc).

- Separar el proceso de "*handshake*" del de intercambio de datos.

- Monitorización de dicha conexión para establecer y guardar ciertos valores (tiempo usado en la conexión, certificado usado, cifrador usado).

- **Conexión TLS con el servidor Apache y gestión de certificados RSA/ECDSA**

Queremos conectar el cliente Android programado con nuestro servidor Apache de forma que establezca una conexión TLS con éste intercambiando certificados RSA o

ECDSA. Para ello necesitamos implementar el navegador de forma que sea capaz de gestionar estos certificados y establezca una conexión segura con el servidor.

Usaremos certificados autofirmados, es decir, actuaremos nosotros como nuestra propia CA. De esta forma, podemos crear los certificados de forma sencilla, los cuales serán suficientes para hacer las pruebas y las medidas que queremos. Para poder gestionar estos certificados autofirmados se ha programado un método dentro de él navegador usando la configuración recomendada que da Google para este tipo de certificados [9]:

```
// Load CAs from an InputStream
// (could be from a resource or ByteArrayInputStream or ...)
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// From https://www.washington.edu/itconnect/security/ca/load-der.crt
InputStream caInput = new BufferedInputStream(new FileInputStream("load-
der.crt"));
Certificate ca;
try {
    ca = cf.generateCertificate(caInput);
    System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
} finally {
    caInput.close();
}
// Create a KeyStore containing our trusted CAs
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);
// Create a TrustManager that trusts the CAs in our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);
// Create an SSLContext that uses our TrustManager
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);
// Tell the URLConnection to use a SocketFactory from our SSLContext
URL url = new URL("https://certs.cac.washington.edu/CAtest/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setSSLSocketFactory(context.getSocketFactory());
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

En este código se crea un contenedor donde alojar los certificados que queremos usar, los cuales carga a través de la forma que le digamos. A partir de ahí crea un *trust manager* que se encarga de autenticar las CA en nuestro contenedor. Después de realizar esto se crea el contexto SSL donde se realizará la conexión segura usando el *trust manager* creado anteriormente. Después se crea una conexión HTTPS que usará un socket del contexto SSL. A partir de aquí tenemos todo preparado para realizar la conexión TLS usando certificados autofirmados.

A partir de este código hemos creado nuestro método que realiza la gestión de los certificados (tanto RSA como ECDSA) y prepara al navegador para realizar la conexión. Nuestro método ha quedado de la siguiente manera:

```
public HttpsURLConnection setHttpsConnection(String urlString, String
certString)throws IOException {

  try {
    CertificateFactory cf = CertificateFactory.getInstance("X.509");
    InputStream caInput = new
BufferedInputStream(context.getAssets().open(certString));
    Certificate ca;
    try {
       ca = cf.generateCertificate(caInput);
        System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
    } finally {
       caInput.close();
    }
    // Create a KeyStore containing our trusted CAs
    String keyStoreType = KeyStore.getDefaultType();
    KeyStore keyStore = KeyStore.getInstance(keyStoreType);
    keyStore.load(null, null);
    keyStore.setCertificateEntry("ca", ca);
    // Create a TrustManager that trusts the CAs in our KeyStore
    String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
    TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
    tmf.init(keyStore);
    // Create an SSLContext that uses our TrustManager
    SSLContext sslcontext = SSLContext.getInstance("TLS");
    sslcontext.init(null, tmf.getTrustManagers(), null);
    HostnameVerifier allHostsValid = new HostnameVerifier() {
        @Override
        public boolean verify(String arg0, SSLSession arg1) {
            return true;
        }
    };

    HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
    // Tell the URLConnection to use a SocketFactory from our SSLContext
    URL url = new URL(urlString);
    HttpsURLConnection urlConnection = (HttpsURLConnection)
url.openConnection();
    urlConnection.setSSLSocketFactory(sslcontext.getSocketFactory());
        return urlConnection;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
  }
```

Para invocar este método y realizar la conexión utilizamos:

```
HttpsURLConnection urlConnection = client.setHttpsConnection(direc, cert);
urlConnection.connect();
urlConnection.disconnect();
```

Y como no hay ningún elemento que pida descarga de datos sólo se realiza el proceso de *handshake* con el servidor estableciendo la conexión TLS con él.

- **Conexiones TLS con servidores comerciales estableciendo los cifradores a usar.**

Para conectarnos con un servidor comercial externo cualquiera necesitamos realizar diferentes operaciones a las hechas anteriormente pues en este caso no manejaremos certificados autofirmados si no que debemos aceptar el que nos envíe el servidor con el que queremos realizar la conexión TLS. Además queremos comprobar si estos servidores realizan conexiones usando certificados ECDSA por lo que tendremos que establecer que sólo se realice esta conexión utilizando cifradores que implementen ECC.

Para ello primero debemos crear un socket para realizar la conexión TLS [18]. A este socket le establecemos los cifradores que permite. Así ha sido implementado todo esto comentado anteriormente:

```
public SSLSocket setConnection(String ciphersuite, String direccion)  {

  try {
    String cipher[] = {ciphersuite};
    SocketFactory socketFactory = SSLSocketFactory.getDefault();
    SSLSocket socket = (SSLSocket) socketFactory.createSocket(direccion, 443);
    socket.setEnabledCipherSuites(cipher);
    HostnameVerifier hostName =
HttpsURLConnection.getDefaultHostnameVerifier();
  return socket;

  } catch (IOException e){
     return null;
  } catch (IllegalArgumentException e){
      return null;
  }
```

Una vez que creamos el socket a usar, nos conectamos de la siguiente manera:

```
SSLSocket socket = client.setConnection(cip, direc);
socket.getSession();
socket.close();
```

Donde creamos el socket con la url del servidor y la suite de cifradores a usar y nos conectamos con el servidor comercial. Con la orden que usamos y al no pedir descargar datos sólo realizamos el proceso de *handshake* con el servidor que es lo que buscamos.

- **Monitorización de dicha conexión**.

Para realizar las diferentes medidas y guardarlas hemos implementado varias líneas de código y métodos que nos ayudarán a realizar dichas mediciones.

Para establecer el tiempo que tarda en realizarse el proceso de handshake hemos usado dos líneas de código colocadas antes y después del momento en el que se establece la conexión. En el primero establecemos la hora en milisegundos justo antes de realizar la conexión y el segundo calcula la hora en milisegundos justo después de realizar la conexión, estableciendo la diferencia entre la primera hora y la segunda, que es el valor de tiempo que ha tardado en realizarse la conexión. Esto lo implementamos de la siguiente manera:

```
long tiempoInicio = System.currentTimeMillis();

urlConnection.connect();

urlConnection.disconnect()

time = Long.toString(System.currentTimeMillis() - tiempoInicio);
```

Para guardar los valores que hemos medido: tiempo, cifradores usados, estado de la conexión, etc; hemos usado un par de métodos que se encargan de crear y escribir archivos .txt con la información que queremos.

El archivo de texto para el servidor programático presenta la siguiente estructura:

```
CERTIFICADO:certificateRSA1024.pem
    - Cipher:  TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    - Tiempo handshake:  42 (ms)
    ----------------------------
    CERTIFICADO:certificateRSA1024.pem
    - Cipher:  TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    - Tiempo handshake:  35 (ms)
    ----------------------------
```

El archivo de texto para los servidores comerciales presenta la siguiente forma:

```
Servidor web:google.com
- Cipher:  TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- ConexiÃ³n:  Error
----------------------------

Servidor web:google.com
- Cipher:  TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- ConexiÃ³n:  Error
----------------------------

Servidor web:google.com
- Cipher:  TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- ConexiÃ³n:  Connect OK
----------------------------

Servidor web:google.com
- Cipher:  TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- ConexiÃ³n:  Connect OK
----------------------------
```

- **Script del lado del servidor**

Para realizar las medidas de tiempos de las conexiones HTTPS desde los navegadores comerciales de Android usaremos una colección de scripts programados para un trabajo similar [19].

Se trata de una colección de scripts CGI (protocolo usado para para realizar una comunicación entre un formulario web y un programa) programados en lenguaje C los cuales tras recibir una cierta URL se encargan de generar una página de respuesta HTML de diferentes tamaños que se envía al usuario cuando termina de ejecutarse el script. A la vez va calculando los tiempos de descarga de dichas páginas HTML, guardando todos los valores en un archivo de texto que presenta la siguiente estructura:

```
Resultados del test para el cliente: Mozilla/5.0 (Linux; Android 4.4.2; bq Aquaris 5 Build/KOT49H) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.93 Mobile Safari/537.36
Cifrador usado: ECDHE-ECDSA-AES128-GCM-SHA256
Tiempo fichero de 1KB = 0.114603 seg
Tiempo fichero de 10KB = 0.119748 seg
Tiempo fichero de 100KB = 0.284543 seg
Tiempo fichero de 1MB = 2.635848 seg

Resultados del test para el cliente: Mozilla/5.0 (Android; Mobile; rv:39.0) Gecko/39.0 Firefox/39.0
Cifrador usado: ECDHE-ECDSA-AES128-GCM-SHA256
Tiempo fichero de 1KB = 0.378486 seg
Tiempo fichero de 10KB = 0.366892 seg
Tiempo fichero de 100KB = 1.629107 seg
Tiempo fichero de 1MB = 3.040988 seg
```

Debido a que los scripts fueron programados hace tiempo no estaban preparados para funcionar en entornos de 64 bits como el que hemos usado en nuestras pruebas, es por ello, que ha habido que realizar diferentes modificaciones a dichos scripts para que funciones correctamente.

Además de lo comentado anteriormente, también hemos tenido que cambiar los valores de las URLs para adaptarlas a nuestro trabajo.

3.2 Metodología de medida

El objetivo de las pruebas es medir diversas características de las conexiones TLS entre el cliente Android programado y los diversos servidores. De esas conexiones nos interesa el tiempo empleado, la energía consumida, cifradores usados y/o soportados. Para ello, hemos establecido una metodología para realizar cada una de las diferentes medidas que queremos realizar.

3.2.1 Medida de tiempo y energía

La medida de tiempo la hemos realizado durante 10 repeticiones para cada una de las medidas para así poder ajustar su valor lo más posible y evitar diferentes fluctuaciones producidas por el entorno de pruebas. Además de la media hemos calculado la desviación típica para hacernos una idea de estas fluctuaciones y por tanto poder establecer hasta qué nivel ese valor nos puede determinar a la hora de sacar conclusiones sobre esos valores medidos. La media y la desviación típica se calcula de la siguiente manera:

- Media:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$ Donde *n* es el número de valores

- Desviación típica:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$ Donde *n* es el número de valores

Para el cálculo de la sobrecarga introducida por diferentes cifradores calcularemos el tanto por ciento correspondiente al valor de la sobrecarga (diferencia entre dos configuraciones) sobre el valor inicial sobre el cual queremos calcular dicha sobrecarga, es decir:

Sobrecarga (segundos) = valor inicial – valor con sobrecarga

Sobrecarga (%)= (sobrecarga (seg) * 100) / valor inicial

Con todos estos valores se han confeccionado tablas donde comparar los diferentes resultados obtenidos.

3.2.2 Medida de conexiones establecidas

Para medir si las conexiones han sido satisfactorias con los valores que les hemos introducido, nos ha bastado con capturar los errores del programa (excepciones) que se producen cuando la conexión no se establece. De este modo cuando se produce alguno de estos problemas, se marca la conexión realizada como error guardando que ha sido una conexión fallida.

Dependiendo de las pruebas estos fallos de conexión nos indican algún problema o nos sirven como valor para saber si con las características dadas se puede realizar la conexión.

Por ejemplo, en las mediciones de tiempo y energía si la conexión da error indica que hay algún problema pues todo se ha programado y configurado para que en este caso siempre se produzca *handshake*. Sin embargo, en el estudio de soporte a ECC de los servidores comerciales un error indica que el cifrador que en ese momento se está probando no es soportado por la web y por tanto ese “error” es un valor válido para nuestro estudio.

Capítulo 4

ESTUDIO DE RENDIMIENTO EN NAVEGADORES COMERCIALES

El objetivo de este capítulo es realizar una comparación entre el uso de cifradores que con ECC con respecto a los cifradores que no los implementan en conexiones entre navegadores comerciales de Android y un servidor Apache configurado para atender dichas conexiones.

En primer lugar se instalaron diferentes navegadores comerciales en el dispositivo Android que usamos en las prueba. El objetivo de ello será establecer la suite de cifradores soportados por todos ellos. Para ello se conectó estos navegadores con el servidor Apache desplegado.

En segundo lugar estudiaremos los tiempos de conexión y descarga de diferentes tamaños de ficheros para cada uno de los navegadores comerciales.

4.1 Soporte ciphersuite en navegadores comerciales

En este apartado estudiaremos los diferentes cifradores de OpenSSL que son soportados por los navegadores comerciales más usados en Android.

4.1.1 Descripción de la prueba

Primero hemos instalado y configurado un servidor Apache en el PC de pruebas en el que también hemos tenido que instalar el módulo de OpenSSL para proporcionar funcionalidades de seguridad.

En primer lugar obtenemos la suite de cifradores soportados por OpenSSL. Para ello usamos el siguiente comando en el terminal: `$>> openssl ciphers –v`

Tras lo cual, obtendremos los cifradores soportados por nuestra versión de OpenSSL y ciertas características de cada uno de ellos. Estas características son:

- Protocolo: protocolo de intercambio de claves. Puede ser TSLv1.2 ó SSLv3.
- Clave: tipo de cifrado de la clave intercambiada en la conexión.
- Autenticación: algoritmo de cifrado usada en la creación de claves.
- Cifrado simétrico: tipo de cifrado simétrico que requiere el cifrador
- MAC (*Message Authentication Code*): algoritmo usado para cifrar el mensaje intercambiado.

En la Tabla 4.1 podemos ver estos valores ordenados.

CIFRADORES	PROTOC.	CLAVE	AUTENT.	CIFRADO SIM.	MAC
ECDHE-RSA-AES256-GCM-SHA384	TLSv1.2	ECDH	RSA	AESGCM(256)	AEAD
ECDHE-ECDSA-AES256-GCM-SHA384	TLSv1.2	ECDH	ECDSA	AESGCM(256)	AEAD
ECDHE-RSA-AES256-SHA384	TLSv1.2	ECDH	RSA	AES(256)	SHA384
ECDHE-ECDSA-AES256-SHA384	TLSv1.2	ECDH	ECDSA	AES(256)	SHA384
ECDHE-RSA-AES256-SHA	SSLv3	ECDH	RSA	AES(256)	SHA1
ECDHE-ECDSA-AES256-SHA	SSLv3	ECDH	ECDSA	AES(256)	SHA1
SRP-DSS-AES-256-CBC-SHA	SSLv3	SRP	DSS	AES(256)	SHA1
SRP-RSA-AES-256-CBC-SHA	SSLv3	SRP	RSA	AES(256)	SHA1
SRP-AES-256-CBC-SHA	SSLv3	SRP	SRP	AES(256)	SHA1
DHE-DSS-AES256-GCM-SHA384	TLSv1.2	DH	DSS	AESGCM(256)	AEAD
DHE-RSA-AES256-GCM-SHA384	TLSv1.2	DH	RSA	AESGCM(256)	AEAD

CIFRADORES	PROTOC.	CLAVE	AUTENT.	CIFRADO SIM.	MAC
DHE-RSA-AES256-SHA256	TLSv1.2	DH	RSA	AES(256)	SHA256
DHE-DSS-AES256-SHA256	TLSv1.2	DH	DSS	AES(256)	SHA256
DHE-RSA-AES256-SHA	SSLv3	DH	RSA	AES(256)	SHA1
DHE-DSS-AES256-SHA	SSLv3	DH	DSS	AES(256)	SHA1
DHE-RSA-CAMELLIA256-SHA	SSLv3	DH	RSA	Camellia(256)	SHA1
DHE-DSS-CAMELLIA256-SHA	SSLv3	DH	DSS	Camellia(256)	SHA1
ECDH-RSA-AES256-GCM-SHA384	TLSv1.2	ECDH/RSA	ECDH	AESGCM(256)	AEAD
ECDH-ECDSA-AES256-GCM-SHA384	TLSv1.2	ECDH/ECDSA	ECDH	AESGCM(256)	AEAD
ECDH-RSA-AES256-SHA384	TLSv1.2	ECDH/RSA	ECDH	AES(256)	SHA384
ECDH-ECDSA-AES256-SHA384	TLSv1.2	ECDH/ECDSA	ECDH	AES(256)	SHA384
ECDH-RSA-AES256-SHA	SSLv3	ECDH/RSA	ECDH	AES(256)	SHA1
ECDH-ECDSA-AES256-SHA	SSLv3	ECDH/ECDSA	ECDH	AES(256)	SHA1
AES256-GCM-SHA384	TLSv1.	RSA	RSA	AESGCM(256)	AEAD
AES256-SHA256	TLSv1.2	RSA	RSA	AES(256)	SHA256
AES256-SHA	SSLv3	RSA	RSA	AES(256)	SHA1
CAMELLIA256-SHA	SSLv3	RSA	RSA	Camellia(256)	SHA1
PSK-AES256-CBC-SHA	SSLv3	PSK	PSK	AES(256)	SHA1
ECDHE-RSA-DES-CBC3-SHA	SSLv3	ECDH	RSA	3DES(168)	SHA1
ECDHE-ECDSA-DES-CBC3-SHA	SSLv3	ECDH	ECDSA	3DES(168)	SHA1
SRP-DSS-3DES-EDE-CBC-SHA	SSLv3	SRP	DSS	3DES(168)	SHA1
SRP-RSA-3DES-EDE-CBC-SHA	SSLv3	SRP	RSA	3DES(168)	SHA1
SRP-3DES-EDE-CBC-SHA	SSLv3	SRP	SRP	3DES(168)	SHA1
EDH-RSA-DES-CBC3-SHA	SSLv3	DH	RSA	3DES(168)	SHA1
EDH-DSS-DES-CBC3-SHA	SSLv3	DH	DSS	3DES(168)	SHA1
ECDH-RSA-DES-CBC3-SHA	SSLv3	ECDH/RSA	ECDH	3DES(168)	SHA1
ECDH-ECDSA-DES-CBC3-SHA	SSLv3	ECDH/ECDSA	ECDH	3DES(168)	SHA1
DES-CBC3-SHA	SSLv3	RSA	RSA	3DES(168)	SHA1
PSK-3DES-EDE-CBC-SHA	SSLv3	PSK	PSK	3DES(168)	SHA1
ECDHE-RSA-AES128-GCM-SHA256	TLSv1.2	ECDH	RSA	AESGCM(128)	AEAD
ECDHE-ECDSA-AES128-GCM-SHA256	TLSv1.2	ECDH	ECDSA	AESGCM(128)	AEAD
ECDHE-RSA-AES128-SHA256	TLSv1.2	ECDH	RSA	AES(128)	SHA256
ECDHE-ECDSA-AES128-SHA256	TLSv1.2	ECDH	ECDSA	AES(128)	SHA256
ECDHE-RSA-AES128-SHA	SSLv3	ECDH	RSA	AES(128)	SHA1
ECDHE-ECDSA-AES128-SHA	SSLv3	ECDH	ECDSA	AES(128)	SHA1
SRP-DSS-AES-128-CBC-SHA	SSLv3	SRP	DSS	AES(128)	SHA1
SRP-RSA-AES-128-CBC-SHA	SSLv3	SRP	RSA	AES(128)	SHA1
SRP-AES-128-CBC-SHA	SSLv3	SRP	SRP	AES(128)	SHA1
DHE-DSS-AES128-GCM-SHA256	TLSv1.2	DH	DSS	AESGCM(128)	AEAD
DHE-RSA-AES128-GCM-SHA256	TLSv1.2	DH	RSA	AESGCM(128)	AEAD
DHE-RSA-AES128-SHA256	TLSv1.2	DH	RSA	AES(128)	SHA256
DHE-DSS-AES128-SHA256	TLSv1.2	DH	DSS	AES(128)	SHA256
DHE-RSA-AES128-SHA	SSLv3	DH	RSA	AES(128)	SHA1
DHE-DSS-AES128-SHA	SSLv3	DH	DSS	AES(128)	SHA1
DHE-RSA-SEED-SHA	SSLv3	DH	RSA	SEED(128)	SHA1
DHE-DSS-SEED-SHA	SSLv3	DH	DSS	SEED(128)	SHA1
DHE-RSA-CAMELLIA128-SHA	SSLv3	DH	RSA	Camellia(128)	SHA1
DHE-DSS-CAMELLIA128-SHA	SSLv3	DH	DSS	Camellia(128)	SHA1
ECDH-RSA-AES128-GCM-SHA256	TLSv1.2	ECDH/RSA	ECDH	AESGCM(128)	AEAD
ECDH-ECDSA-AES128-GCM-SHA256	TLSv1.2	ECDH/ECDSA	ECDH	AESGCM(128)	AEAD
ECDH-RSA-AES128-SHA256	TLSv1.2	ECDH/RSA	ECDH	AES(128)	SHA256

CIFRADORES	PROTOC.	CLAVE	AUTENT.	CIFRADO SIM.	MAC
ECDH-ECDSA-AES128-SHA256	TLSv1.2	ECDH/ECDSA	ECDH	AES(128)	SHA256
ECDH-RSA-AES128-SHA	SSLv3	ECDH/RSA	ECDH	AES(128)	SHA1
ECDH-ECDSA-AES128-SHA	SSLv3	ECDH/ECDSA	ECDH	AES(128)	SHA1
AES128-GCM-SHA256	TLSv1.2	RSA	RSA	AESGCM(128)	AEAD
AES128-SHA256	TLSv1.2	RSA	RSA	AES(128)	SHA256
AES128-SHA	SSLv3	RSA	RSA	AES(128)	SHA1
SEED-SHA	SSLv3	RSA	RSA	SEED(128)	SHA1
CAMELLIA128-SHA	SSLv3	RSA	RSA	Camellia(128)	SHA1
PSK-AES128-CBC-SHA	SSLv3	PSK	PSK	AES(128)	SHA1
ECDHE-RSA-RC4-SHA	SSLv3	ECDH	RSA	RC4(128)	SHA1
ECDHE-ECDSA-RC4-SHA	SSLv3	ECDH	ECDSA	RC4(128)	SHA1
ECDH-RSA-RC4-SHA	SSLv3	ECDH/RSA	ECDH	RC4(128)	SHA1
ECDH-ECDSA-RC4-SHA	SSLv3	ECDH/ECDSA	ECDH	RC4(128)	SHA1
RC4-SHA	SSLv3	RSA	RSA	RC4(128)	SHA1
RC4-MD5	SSLv3	RSA	RSA	RC4(128)	MD5
PSK-RC4-SHA	SSLv3	PSK	PSK	RC4(128)	SHA1
EDH-RSA-DES-CBC-SHA	SSLv3	DH	RSA	DES(56)	SHA1
EDH-DSS-DES-CBC-SHA	SSLv3	DH	DSS	DES(56)	SHA1
DES-CBC-SHA	SSLv3	RSA	RSA	DES(56)	SHA1

Tabla 4.1: Cifradores para conexiones TLS/SSL en la versión 1.0.1f de OpenSSL.

Esta lista de cifradores es la que se ha usado para probar el soporte de diferentes navegadores de Android. Para ello, hemos ido modificando la directiva `SSLCipherSuite` para establecer cada uno de los cifradores cuyo soporte queremos probar.

Se ha probado con el navegador por defecto de Android (Chrome, desde la versión 4.4 de Android), Firefox, Dolphin y Opera.

El servidor Apache se configuró para permitir conexiones SSL/TLS descomentando la directiva `SSL` que habilita este modo en Apache y activando el módulo OpenSSL mediante los siguientes comandos en el terminal de Linux:

```
$>> sudo a2ensite default-ssl.conf

$>> sudo a2enmodssl
```

A continuación se ha modificó el archivo de configuración de ssl *default-ssl.conf* añadiendo las siguientes directivas que permitían ir estableciendo el cifrador a probar:

```
SSLProtocol all -SSLv2 -SSLv3
SSLHonorCipherOrder     on
SSLCompression          off
SSLCipherSuite <cifrador o cifradores separados por: >
```

Para probar el soporte a los cifradores nos hemos conectado a la página por defecto de Apache para ver si se ha establecido la conexión con éxito. Como hemos comentado anteriormente, mediante la directiva `SSLCipherSuite` fuimos cambiando los cifradores en el archivo de configuración SSL de Apache para ir probando uno a uno con cada navegador. A su vez fuímos cambiando el certificado usado dependiendo de si usamos un cifrador que requería certificados RSA o ECDSA.

4.1.2 Soporte ciphersuite

Tras realizar las pruebas de la forma comentada anteriormente obtenemos los *ciphersuites* de OpenSSL soportados por cada navegador, los cuales podemos ver en la Tabla 4.2. En dicha tabla están resaltados aquellos cifradores que comparten los cuatro navegadores.

Cifrador	Chrome	Firefox	Dolphin	Opera
ECDHE-RSA-AES256-GCM-SHA384	x	x	x	x
ECDHE-ECDSA-AES256-GCM-SHA384	x	x	x	x
ECDHE-RSA-AES256-SHA384	x	x	x	x
ECDHE-ECDSA-AES256-SHA384	x	x	x	x
ECDHE-RSA-AES256-SHA	SI	SI	SI	SI
ECDHE-ECDSA-AES256-SHA	SI	SI	SI	SI
SRP-DSS-AES-256-CBC-SHA	x	x	x	x
SRP-RSA-AES-256-CBC-SHA	x	x	x	x
SRP-AES-256-CBC-SHA	x	x	x	x
DHE-DSS-AES256-GCM-SHA384	x	x	x	x
DHE-RSA-AES256-GCM-SHA384	x	x	x	x
DHE-RSA-AES256-SHA256	x	x	x	x
DHE-DSS-AES256-SHA256	x	x	x	x
DHE-RSA-AES256-SHA	SI	SI	SI	SI
DHE-DSS-AES256-SHA	x	x	x	x
DHE-RSA-CAMELLIA256-SHA	x	x	x	x
DHE-DSS-CAMELLIA256-SHA	x	x	x	x
ECDH-RSA-AES256-GCM-SHA384	x	x	x	x
ECDH-ECDSA-AES256-GCM-SHA384	x	x	x	x

Cifrador	Chrome	Firefox	Dolphin	Opera
ECDH-RSA-AES256-SHA384	x	x	x	x
ECDH-ECDSA-AES256-SHA384	x	x	x	x
ECDH-RSA-AES256-SHA	x	x	x	x
ECDH-ECDSA-AES256-SHA	x	x	SI	x
AES256-GCM-SHA384	x	x	x	x
AES256-SHA256	x	x	x	x
AES256-SHA	SI	SI	SI	SI
CAMELLIA256-SHA	x	x	x	x
PSK-AES256-CBC-SHA	x	x	x	x
ECDHE-RSA-DES-CBC3-SHA	x	x	SI	x
ECDHE-ECDSA-DES-CBC3-SHA	x	x	SI	x
SRP-DSS-3DES-EDE-CBC-SHA	x	x	x	x
SRP-RSA-3DES-EDE-CBC-SHA	x	x	x	x
SRP-3DES-EDE-CBC-SHA	x	x	x	x
EDH-RSA-DES-CBC3-SHA	x	x	SI	x
EDH-DSS-DES-CBC3-SHA	x	x	x	x
ECDH-RSA-DES-CBC3-SHA	x	x	x	x
ECDH-ECDSA-DES-CBC3-SHA	x	x	SI	x
DES-CBC3-SHA	SI	SI	SI	SI
PSK-3DES-EDE-CBC-SHA	x	x	x	x
ECDHE-RSA-AES128-GCM-SHA256	SI	SI	x	SI
ECDHE-ECDSA-AES128-GCM-SHA256	SI	SI	SI	SI
ECDHE-RSA-AES128-SHA256	x	x	x	x
ECDHE-ECDSA-AES128-SHA256	x	x	x	x
ECDHE-RSA-AES128-SHA	SI	SI	SI	SI
ECDHE-ECDSA-AES128-SHA	SI	SI	SI	SI
SRP-DSS-AES-128-CBC-SHA	x	x	x	x
SRP-RSA-AES-128-CBC-SHA	x	x	x	x
SRP-AES-128-CBC-SHA	x	x	x	x
DHE-DSS-AES128-GCM-SHA256	x	x	x	x
DHE-RSA-AES128-GCM-SHA256	v	x	x	SI
DHE-RSA-AES128-SHA256	x	x	x	x
DHE-DSS-AES128-SHA256	x	x	x	x
DHE-RSA-AES128-SHA	SI	SI	SI	SI
DHE-DSS-AES128-SHA	x	x	x	x
DHE-RSA-SEED-SHA	x	x	x	x
DHE-DSS-SEED-SHA	x	x	x	x
DHE-RSA-CAMELLIA128-SHA	x	x	x	x
DHE-DSS-CAMELLIA128-SHA	x	x	x	x
ECDH-RSA-AES128-GCM-SHA256	x	x	x	x
ECDH-ECDSA-AES128-GCM-SHA256	x	x	x	x
ECDH-RSA-AES128-SHA256	x	x	x	x
ECDH-ECDSA-AES128-SHA256	x	x	x	x
ECDH-RSA-AES128-SHA	x	x	x	x
ECDH-ECDSA-AES128-SHA	x	x	v	x
AES128-GCM-SHA256	v	x	x	v
AES128-SHA256	x	x	x	x
AES128-SHA	SI	SI	SI	SI
SEED-SHA	x	x	x	x
CAMELLIA128-SHA	x	x	x	x
PSK-AES128-CBC-SHA	x	x	x	x
ECDHE-RSA-RC4-SHA	SI	x	SI	SI

Cifrador	Chrome	Firefox	Dolphin	Opera
ECDH-RSA-RC4-SHA	x	x	x	x
ECDH-ECDSA-RC4-SHA	x	x	SI	x
RC4-SHA	SI	x	SI	SI
RC4-MD5	SI	x	SI	SI
PSK-RC4-SHA	x	x	x	x
EDH-RSA-DES-CBC-SHA	x	x	x	x
EDH-DSS-DES-CBC-SHA	x	x	x	x
DES-CBC-SHA	x	x	x	x

Tabla 4.2. Cifradores soportados por navegadores comerciales

4.2 Medidas de tiempo en navegadores comerciales

El objetivo de estas pruebas ha sido evaluar el comportamiento de los navegadores comerciales de Android ante conexiones HTTPS usando diferentes cifradores. Nos interesa sobre todo el comportamiento de los cifradores que usen ECC con respecto al resto.

4.2.1 Descripción de la prueba

Para la realización de la prueba hemos creado una red aislada del exterior con la siguiente configuración:

Servidor web ↔ Router ↔ Dispositivo Android.

Realizamos la medida de eficiencia en coste temporal mediante una aplicación CGI alojada en el servidor que mide el tiempo de descarga de ficheros de diferentes tamaños: 1KB, 10KB, 100KB y 1MB.

Para que el servidor Apache soporte CGI debemos añadir en el archivo *000-default.conf* de configuración las siguientes directivas:

```
#LoadModule cgi_module /usr/lib/apache2/modules/mod_cgi.so
#enables all user cgi's
<DirectoryMatch ./cgi-bin>
        Options ExecCGI
        SetHandler cgi-script
</DirectoryMatch>

<Directory ./cgi-bin>
        Options ExecCGI
        SetHandler cgi-script
</Directory>
```

Tras esto, para ejecutar los scripts basta con introducir la siguiente url en los navegadores a probar:

http://IP:puerto/cgi-bin/cgimitest.cgi?1

Al ejecutar estos scripts, los navegadores requieren que soporten varias ventanas emergentes por lo que no hemos podido realizar las pruebas en todos los navegadores usados en la prueba anterior, pues el navegador Dolphin no admite más de tres ventanas emergentes. Debido a esto, se han realizado las medidas usando el navegador nativo de Android (Chrome), Firefox y Opera, que sí cumplen los requisitos necesarios.

Las medidas han consistido en el establecimiento de conexiones seguras HTTPS. Para ello se han empleado los cifradores comunes a los tres navegadores, forzando la configuración del servidor Apache a que sólo acepte cada cifrador específico que queríamos usar. Se modificó el archivo *default-ssl.conf* para cada cifrador mediante la directiva `SSLCipherSuite,` que establece los cifradores a usar. En la Tabla 4.2 del apartado anterior podemos ver resaltados los cifradores que hemos usado.

4.2.2 Medidas de tiempo

Tras la realización de las medidas hemos obtenido los siguientes resultados en promedio. Para realizar una comparativa más completa entre los tres navegadores se han extraído los resultados en función del tamaño de fichero para así visualizar de una forma más cómoda el estudio. En las tablas de resultados se han resaltado los cifradores que usan ECDHE y/o ECDSA pues son los que estudiaremos más detalladamente.

- **Fichero 1KB**

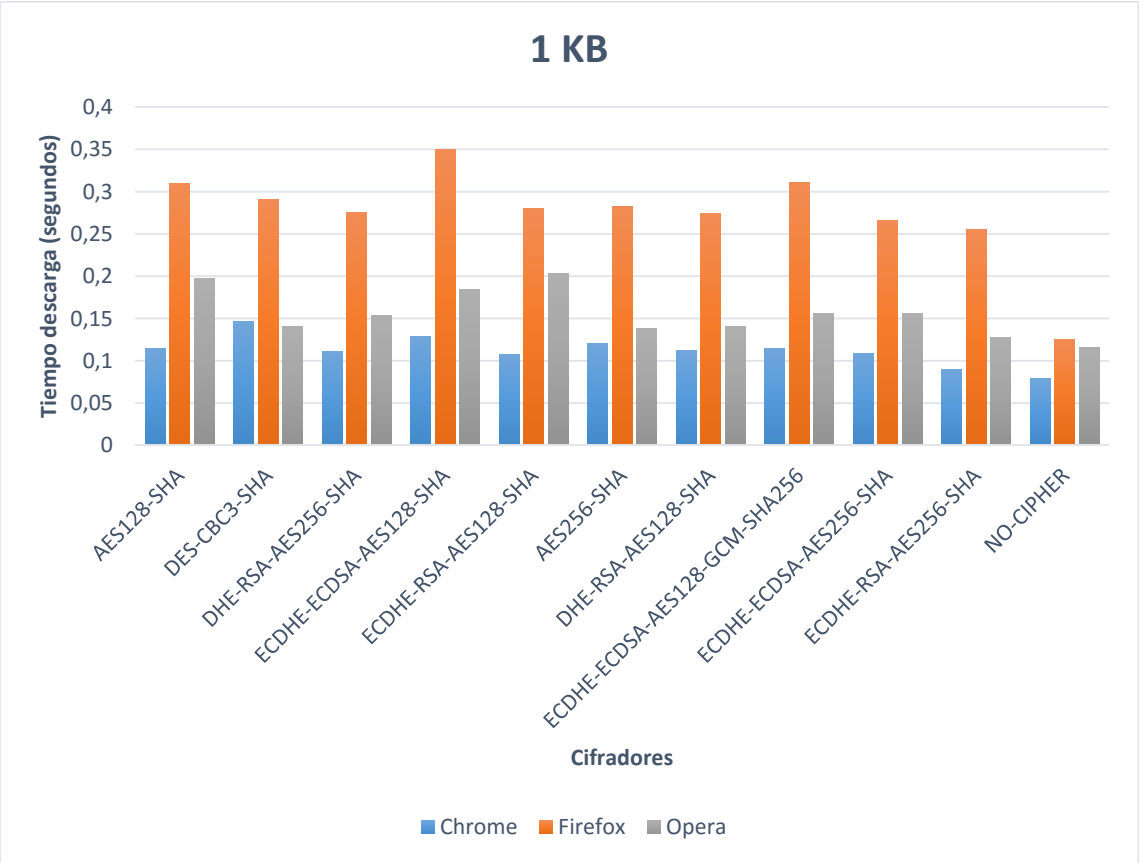


Figura 4.1. Medidas de tiempo para 1K

En la Figura 4.1 podemos ver que el tiempo obtenido por el navegador Firefox es mucho mayor al obtenido por Chrome y Opera. Entre estos dos las diferencias de tiempo vemos que es muy pequeña, teniendo un valor similar, aunque siendo mayor en el caso de Opera. Por ello, para archivos de tamaña 1K podemos decir que le navegador más rápido en Chrome.

Con respecto a los cifradores probados, comprobamos que evidentemente al no usar cifradores obtenemos valores de tiempo menor. El cifrador ECDHE-RSA-AES256-SHA es el que menos sobrecarga introduce en los tres navegadores, tal y como podemos ver en la Tabla 4.3.

CIPHER	Chrome (seg)	Chrome (%)	Firefox (seg)	Firefox (%)	Opera (seg)	Opera (%)
AES128-SHA	0.0361	45.79	0.1851	148.12	0.0817	70.48
DES-CBC3-SHA	0.0670	84.91	0.1662	133.00	0.0241	20.82
DHE-RSA-AES256-SHA	0.0323	40.96	0.1501	120.15	0.0376	32.39
ECDHE-ECDSA-AES128-SHA	0.0502	63.54	0.2247	179.86	0.0683	58.87
ECDHE-RSA-AES128-SHA	0.0289	36.65	0.1550	124.10	0.0869	74.97
AES256-SHA	0.0416	52.76	0.1579	126.37	0.0227	19.56
DHE-RSA-AES128-SHA	0.0328	41.52	0.1487	119.04	0.0240	20.77
ECDHE-ECDSA-AES128-GCM-SHA256	0.0359	45.48	0.1860	148.84	0.0405	34.96
ECDHE-ECDSA-AES256-SHA	0.0299	37.83	0.1415	113.24	0.0399	34.44
ECDHE-RSA-AES256-SHA	0.0110	13.99	0.1298	103.88	0.0116	10.04

Tabla 4.3. Sobrecarga en descarga de archivo de 1KB

Vemos que Chrome promedia una sobrecarga de 46% y Opera de un 37%, mientras que Firefox de 132%, siendo mucho mayor que en los dos anteriores. En los tres casos el cifrador ECDHE-RSA-AES256-SHA introduce una sobrecarga menor siendo el más eficiente de todos los probados en la descarga de un archivo de 1K.

Con respecto a los cifradores que implementan ECDSA vemos que presentan una sobrecarga mayor a su equivalente RSA.

- **Fichero 10KB**

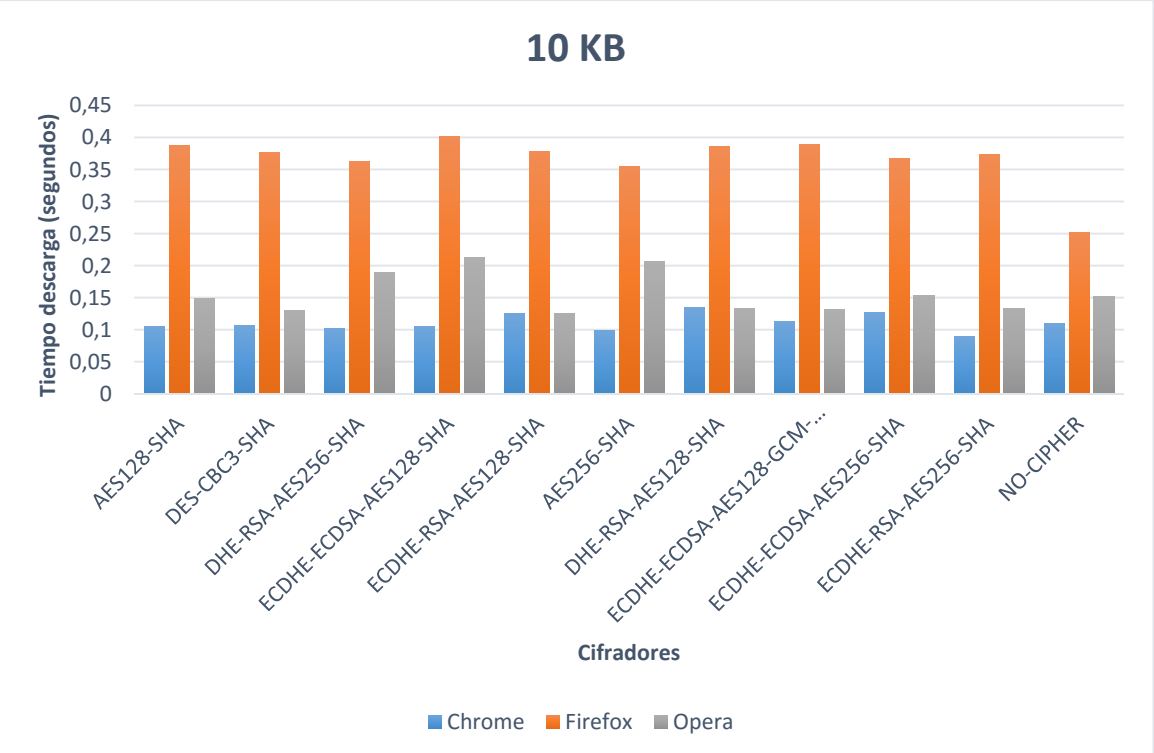


Figura 4.2. Medidas de tiempo para 10K

Para un archivo de 10K comprobamos en la Figura 4.2 que Firefox presenta un consumo temporal similar para todos los cifradores probados. Siendo el cifrador AES256-SHA el que menos tiempo consume en Firefox. Para Chrome y Opera es el cifrador ECDHE-RSA-AES256-SHA el más rápido

CIPHER	Chrome (seg)	Chrome (%)	Firefox (seg)	Firefox (%)	Opera (seg)	Opera (%)
AES128-SHA	0.0041	4.17	0.1359	54.21	0.0158	11.90
DES-CBC3-SHA	0.0067	6.68	0.1261	50.28	-0.0026	-1.93
DHE-RSA-AES256-SHA	0.0024	2.37	0.1123	44.76	0.0562	42.50
ECDHE-ECDSA-AES128-SHA	0.0044	4.43	0.1513	60.34	0.0797	60.25
ECDHE-RSA-AES128-SHA	0.0249	24.90	0.1274	50.79	-0.0065	-4.94
AES256-SHA	-0.0007	-0.74	0.1032	41.14	0.0741	55.97
DHE-RSA-AES128-SHA	0.0341	34.07	0.1344	53.59	0.0011	0.83
ECDHE-ECDSA-AES128-GCM-SHA256	0.0127	12.74	0.1383	55.15	-0.0011	-0.85
ECDHE-ECDSA-AES256-SHA	0.0268	26.78	0.1167	46.52	0.0216	16.33
ECDHE-RSA-AES256-SHA	-0.0109	-10.89	0.1232	49.12	0.0011	0.80

Tabla 4.4. Sobrecarga en descarga de archivo de 10KB

En la Tabla 4.4 vemos que Chrome promedia una sobrecarga de un 10% y Opera de un 18%, mientras que Firefox presenta una sobrecarga mayor, de un 50%.

Para Chrome el cifrador ECDHE-RSA-AES256-SHA es el que menos sobrecarga presenta. Para Opera es el ECDHE-RSA-AES128-SHA. Y para Firefox el AES256-SHA.

Con respecto a los cifradores con ECDSA vemos que en este caso si comparamos ECDHE-ECDSA-AES128-SHA y ECDHE-RSA-AES128-SHA, para Chrome el primero presenta menos sobrecarga mientras que para los otros dos navegadores es el segundo el que nos da mejor respuesta. Si la comparación la realizamos entre ECDHE-ECDSA-AES256-SHA y ECDHE-RSA-AES256-SHA, comprobamos que el para Firefox la sobrecarga que introducen es similar, mientras que para Chrome y Opera ECDHE-RSA-AES256-SHA ofrece una sobrecarga mucho menor.

En este caso vemos que para varios cifradores la sobrecarga presenta un valor negativo, de lo que podemos concluir que teniendo en cuenta posibles variaciones en las medidas debido al entorno usado la sobrecarga en estos casos es muy similar al valor de tiempo al no usar ningún cifrador. Esto es lógico que ocurra conforme aumentemos el valor del fichero a descargar pues aumenta el tiempo empleado y por tanto, disminuye la importancia del valor de tiempo usado en realizar la conexión segura sobre el tiempo total.

- **Fichero 100KB**

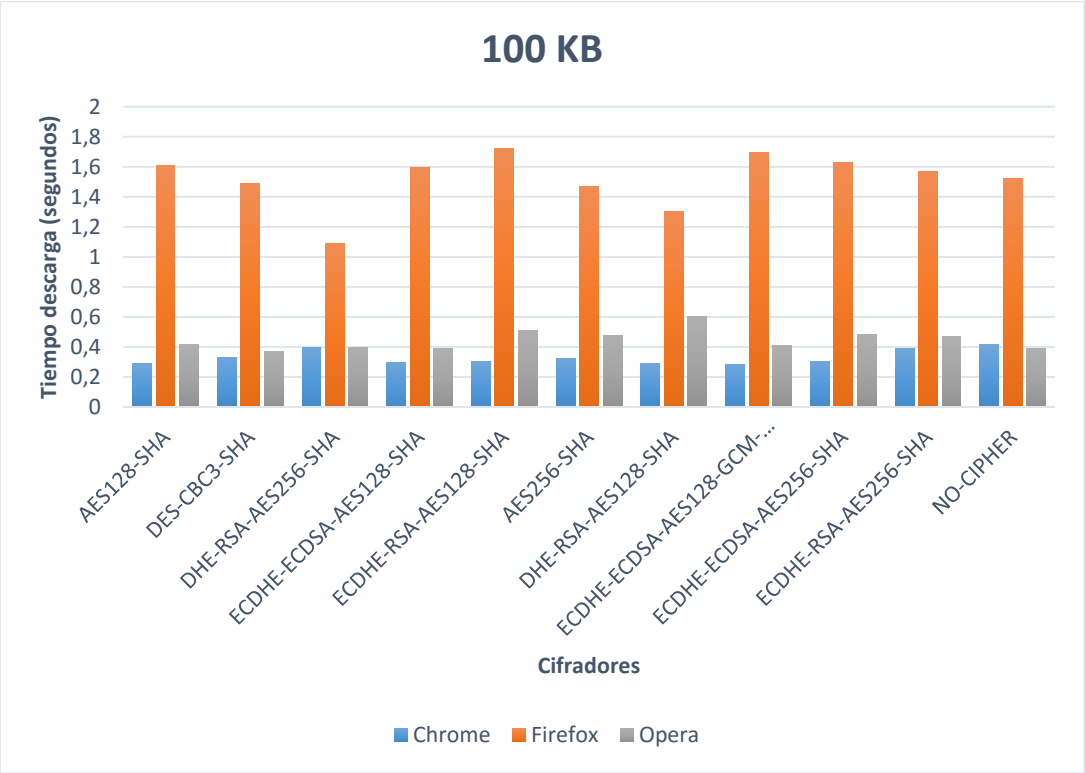


Figura 4.3. Medidas de tiempo para 100K

Viendo la Figura 4.3 comprobamos que para un archivo de 100K el tiempo de descarga para Firefox es mucho mayor que para Chrome y Opera, pues estos dos presentan un comportamiento muy similar para todos los cifradores y en el caso de no usar ninguno, siendo siempre el valor de tiempo para Chrome un poco menor que el de Opera.

CIPHER	Chrome (seg)	Chrome (%)	Firefox (seg)	Firefox (%)	Opera (seg)	Opera (%)
AES128-SHA	-0.0072	-2.43	0.2754	20.66	0.0267	6.88
DES-CBC3-SHA	0.0360	12.19	0.1581	11.86	-0.0205	-5.27
DHE-RSA-AES256-SHA	0.0983	33.27	-0.2443	-18.33	0.0101	2.61
ECDHE-ECDSA-AES128-SHA	0.0039	1.30	0.2638	19.79	0.0011	0.27
ECDHE-RSA-AES128-SHA	0.0068	2.33	0.3895	29.22	0.1184	30.44
AES256-SHA	0.0295	9.97	0.1361	10.21	0.0902	23.21
DHE-RSA-AES128-SHA	-0.0044	-1.48	-0.0311	-2.34	0.2140	55.02
ECDHE-ECDSA-AES128-GCM-SHA256	-0.0087	-2.93	0.3616	27.12	0.0222	5.71
ECDHE-ECDSA-AES256-SHA	0.0090	3.06	0.2946	22.10	0.0945	24.30
ECDHE-RSA-AES256-SHA	0.0910	30.81	0.2322	17.42	0.0820	21.10

Tabla 4.5. Sobrecarga en descarga de archivo de 100KB

En la Tabla 4.5 vemos los valores de sobrecarga introducida por los cifradores en la descarga de un archivo de 100K. Comprobamos que en este caso Chrome promedia una sobrecarga menor (8%), seguido por Firefox (14%) y siendo Opera el que introduce un valor mayor (16%).

Con respecto a ECDSA, vemos que en este caso los cifradores que lo implementan presentan un mejor comportamiento en Chrome y Firefox que los que usan RSA. En el caso de Opera, se observa que para ECDHE-ECDSA-AES128-SHA la sobrecarga introducida es mucho menor que la introducida por su equivalente, ECDHE-RSA-AES128-SHA.

Al igual que en el caso anterior, tenemos valores de sobrecarga negativos debido a posibles fluctuaciones en las medidas y a la disminución de la importancia de la sobrecarga sobre el valor total de tiempo de descarga.

- **Fichero 1MB**

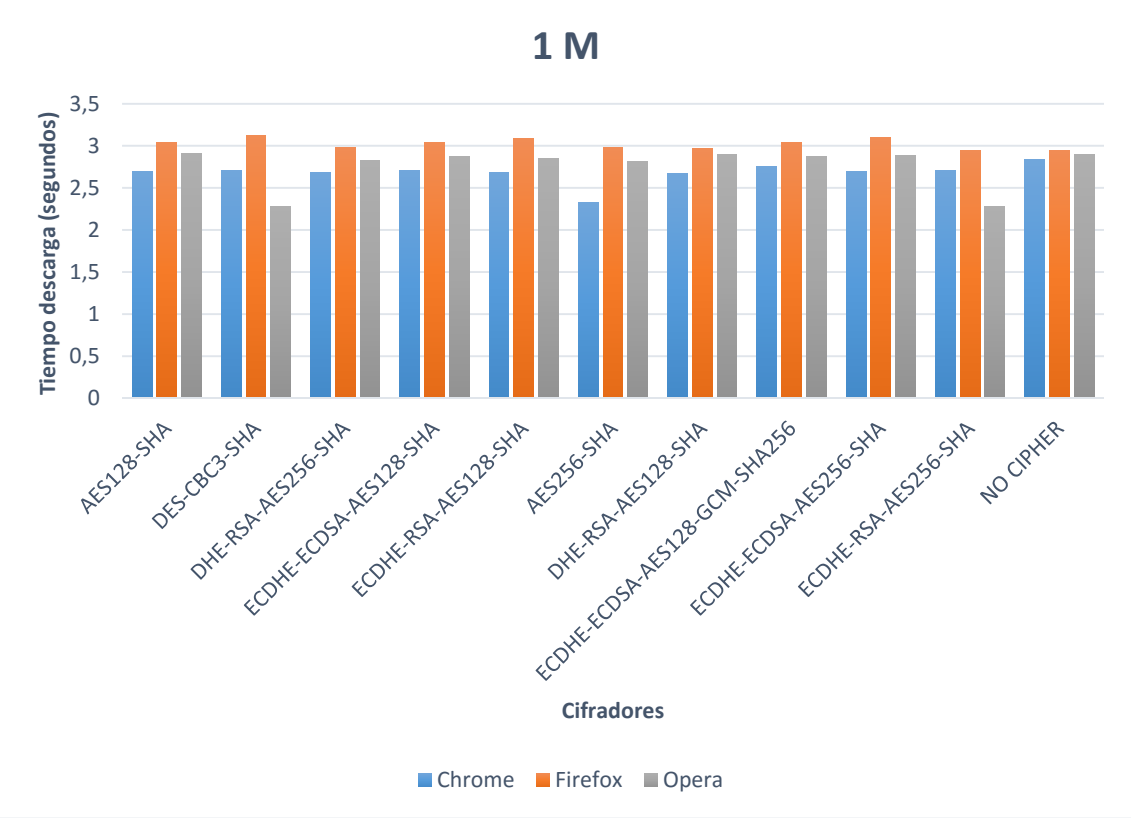


Figura 4.4. Medidas de tiempo para 1M

Observando la Figura 4.4 vemos que en este caso los valores de tiempo de descarga de los tres navegadores es más cercano, siendo como en los casos anteriores, mayores en el caso del navegador Firefox.

Para Chrome el cifrador más rápido es AES256-SHA, mientras que para Firefox y Opera se trata del ECDHE-RSA-AES256-SHA.

CIPHER	Chrome (seg)	Chrome (%)	Firefox (seg)	Firefox (%)	Opera (seg)	Opera (%)
AES128-SHA	0.1747	6.91	0.0927	3.15	0.2718	10.33
DES-CBC3-SHA	0.1800	7.13	0.1766	6.00	-0.3476	-13.21
DHE-RSA-AES256-SHA	0.1541	6.10	0.0326	1.11	0.1932	7.34
ECDHE-ECDSA-AES128-SHA	0.1771	7.01	0.0931	3.17	0.2393	9.09
ECDHE-RSA-AES128-SHA	0.1522	6.02	0.1463	4.97	0.2188	8.31
AES256-SHA	-0.2055	-8.13	0.0380	1.29	0.1870	7.10
DHE-RSA-AES128-SHA	0.1435	5.68	0.0278	0.94	0.26445	10.05
ECDHE-ECDSA-AES128-GCM-SHA256	0.2271	8.99	0.0982	3.34	0.2424	9.21
ECDHE-ECDSA-AES256-SHA	0.1636	6.48	0.1514	5.14	0.2510	9.54
ECDHE-RSA-AES256-SHA	0.1771	7.01	0.0052	0.18	-0.3470	-13.18

Tabla 4.6. Sobrecarga en descarga de archivo de 1MB

Si observamos la Tabla 4.6 vemos que en promedio Firefox es el que ofrece una sobrecarga menor (3%) que Chrome (5%) y Opera (4%).

En el caso de cifradores que implementan ECDSA vemos que ofrecen valores similares de sobrecarga a su complementario RSA excepto en el caso de ECDHE-ECDSA-AES256-SHA que ofrece valores mayores de sobrecarga para Firefox y Opera que su complementario ECDHE-RSA-AES256-SHA.

Como en los casos anteriores, hemos obtenido algunos valores negativos de sobrecarga al realizar las medidas.

4.3 Discusión

En este apartado usaremos los resultados obtenidos anteriormente para compararlos con el obtenido en un estudio parecido realizado anteriormente. El objetivo de esto será establecer el soporte actual de ECC.

También se sacarán conclusiones de rendimiento entre el uso de ECDHE con respecto al resto de cifradores y de RSA con respecto a ECDSA en las conexiones seguras realizadas mediante navegadores comerciales. Con ello se pretende averiguar si el uso de uno con respecto a otro es más eficiente en valores de tiempo en su uso por los navegadores comerciales de Android.

4.3.1 Validación de resultados

Comparamos con los resultados obtenidos en las pruebas descritas en este capítulo con un estudio anterior [1] sobre OpenSSL v 0.9.8 y una versión anterior de Android.

Los cifradores soportados por OpenSSL en el estudio citado podemos verlos en Tabla 4.7. Comprobamos que en la versión 1.0.1f de OpenSSL ha aumentado considerablemente, especialmente con respecto a los cifradores que usan ECDHE y ECDSA, los cuales no eran soportados en el momento de realizar dicho estudio.

CIPHER	Prot.	Intercambio Clave	Autent.	Cifrado	Mac
DHE-RSA-AES256-SHA	SSLv3	DH	RSA	AES(256)	SHA1
DHE-DSS-AES256-SHA	SSLv3	DH	DSS	AES(256)	SHA1
AES256-SHA	SSLv3	RSA	RSA	AES(256)	SHA1
EDH-RSA-DES-CBC3-SHA	SSLv3	DH	RSA	3DES(168)	SHA1
EDH-DSS-DES-CBC3-SHA	SSLv3	DH	DSS	3DES(168)	SHA1
DES-CBC3-SHA	SSLv3	RSA	RSA	3DES(168)	SHA1
DES-CBC3-MD5	SSLv2	RSA	RSA	3DES(168)	MD5
DHE-RSA-AES128-SHA	SSLv3	DH	RSA	AES(128)	SHA1
DHE-DSS-AES128-SHA	SSLv3	DH	DSS	AES(128)	SHA1
AES128-SHA	SSLv3	RSA	RSA	AES(128)	SHA1
RC2-CBC-MD5	SSLv2	RSA	RSA	RC2(128)	MD5
RC4-SHA	SSLv3	RSA	RSA	RC4(128)	SHA1
RC4-MD5	SSLv3	RSA	RSA	RC4(128)	MD5
RC4-MD5	SSLv2	RSA	RSA	RC4(128)	MD5
EDH-RSA-DES-CBC-SHA	SSLv3	DH	RSA	DES(56)	SHA1
EDH-DSS-DES-CBC-SHA	SSLv3	DH	DSS	DES(56)	SHA1
DES-CBC-SHA	SSLv3	RSA	RSA	DES(56)	SHA1
DES-CBC-MD5	SSLv2	RSA	RSA	DES(56)	MD5
EXP-EDH-RSA-DES-CBC-SHA	SSLv3	DH(512)	RSA	DES(40)	SHA1
EXP-EDH-DSS-DES-CBC-SHA	SSLv3	DH(512)	DSS	DES(40)	SHA1
EXP-DES-CBC-SHA	SSLv3	RSA(512)	RSA	DES(40)	SHA1
EXP-RC2-CBC-MD5	SSLv3	RSA(512)	RSA	RC2(40)	MD5
EXP-RC4-MD5	SSLv3	RSA(512)	RSA	RC4(40)	MD5

Tabla 4.7. Cifradores soportados por OpenSSL v 0.9.8

Con ello vemos que el soporte de ECC ha aumentado, pasando de no ser usado a ser uno los implementados.

Para ver un ejemplo, en la Tabla 4.7 tenemos resaltados los cifradores soportados por en el navegador Firefox en el momento de la realización de las pruebas del estudio referenciado. Comprobamos cómo en la versión de Android que estamos

usando en este proyecto (Android 4.4) y por tanto, en los navegadores programados para su uso en los dispositivos que lo llevan, se ha implementado el uso de ECC para realizar conexiones seguras, ya sea mediante ECDHE, ECDSA o ambos. En el ejemplo que estamos estudiando para el navegador Firefox, los cifradores añadidos en su implementación que usan ECC los podemos ver en la Tabla 4.8.

CIPHER
ECDHE-ECDSA-AES128-SHA
ECDHE-RSA-AES128-SHA
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES256-SHA
ECDHE-RSA-AES256-SHA

Tabla 4.8. Cifradores que implementan ECC usados por Firefox.

4.4.2 Rendimiento de ECC en navegadores comerciales

Tras las medidas de tiempo de conexiones seguras realizadas mediante los navegadores comerciales Firefox, Chrome y Opera podemos concluir:

- La implementación de ECC en servidores, en nuestro caso Apache (y su módulo OpenSSL), es muy completa y permite la elección de diferentes cifradores según uso o requerimientos del sistema o red.

- La implementación de ECC en los cifradores usados en conexiones TSL en dispositivos móviles, y en especial en Android ha aumentado en las últimas versiones del sistema operativo, teniendo una gran variedad de ellos. Permitiendo combinar su uso con métodos tradicionales como en el caso de los cifradores ECDHE-RSA.

- En general, los cifradores que usan ECDHE y por tanto, requieren el uso de ECC para realizar el intercambio de llaves (RSA o ECDSA) son más rápidos que el resto.

- En el caso del uso de llaves ECDSA, no queda claro que su uso sea más eficiente pues en la mayoría de los casos RSA es más rápido o tiene un valor similar de tiempo al del uso de ECDSA.

En conclusión, podemos decir que la inclusión y uso de ECC en las conexiones TSL no sólo es más eficiente y sobre todo más seguro según la teoría, sino que también lo son en la práctica, sobre todo a la hora del cifrado del canal de intercambio. En el caso de las claves ECDSA no podemos llegar a una conclusión clara con estas pruebas por lo que las estudiaremos con más detalle en el siguiente capítulo donde nos centraremos en su uso para estudiar su eficiencia con respecto a las claves RSA.

Capítulo 5

ESTUDIO DE RENDIMIENTO EN NAVEGADOR PROGRAMÁTICO

El objetivo de estas pruebas es realizar una comparación entre el uso de certificados que usa ECC (a partir de ahora certificados ECDSA) con respecto al tipo de certificados más usados en la actualidad, los certificados RSA. Con ello queremos establecer si las ventajas que a priori tiene el uso de certificados ECDSA en las conexiones móviles (en nuestro caso Android) se cumplen en un entorno real.

Para la realización de dichas medidas nos interesa el momento del *handshake*, es decir, el momento en el que se realiza autenticación para el intercambio de claves de sesión.

Intentaremos hallar la diferencia en coste temporal/energético entre el uso de ECC y RSA con diferentes tamaños de clave.

Finalmente estudiaremos el soporte de los servidores comerciales más visitados a dichos certificados.

5.1 Medida de eficiencia en conexiones con el servidor Apache

Para la realización de las pruebas de tiempo y energía se programó un cliente Android que realizara conexiones seguras mediante TLS con el servidor Apache configurado usando certificados RSA y ECDSA. Este cliente separa el proceso de *handshake* del de intercambio de datos, realizándose sólo el primero. La duración de tiempo de este proceso y la energía consumida durante él es lo que queremos medir con esta prueba.

Para las medidas de energía usamos una aplicación externa llamada *PowerTutor* creada por la Universidad de Michigan [15]. Dicha aplicación se encarga, entre otras cosas, de calcular la energía gastada por cada una de las aplicaciones que se encuentran en funcionamiento en el periodo de tiempo que *PowerTutor* mide.

En el capítulo 3 se explica con detalle cómo ha sido desarrollado el cliente Android usado y las características de PowerTutor.

Además del cliente Android, para realizar las conexiones TLS con el servidor Apache se necesitan los certificados RSA y ECDSA que queremos probar. A continuación se explica cómo fueron generados.

5.1.1 Soporte certificados RSA y ECDSA

Las pruebas las realizaremos con diferentes tipos de certificados RSA y ECDSA con tamaños de clave diferente. La idea es compararlos con tamaños de clave equivalentes. En capítulos anteriores se ha explicado cómo ECDSA necesita menos bits de clave para ofrecer una seguridad equivalente a RSA. Por ello se comparan los certificados que ofrecen un equivalente nivel de seguridad pues de esta forma podremos averiguar si el ahorro en bits que ofrece ECC a la hora de cifrar compensa en términos de tiempo y energía al que ofrece RSA.

En la Tabla 5.1 podemos ver dichas equivalencias cuyos valores usaremos para generar los certificados.

DH/DSA/RSA (bits)	ECC(ECDSA) (bits)
1024	160
2048	224
3072	256
7680	384
15360	512

Tabla 5.1. Equivalencias de RSA y ECDSA [4]

Por motivos de eficiencia y facilidad de uso, los certificados los crearemos autofirmados, es decir, seremos nuestra propia CA (*Certificate Authority*), o lo que es lo mismo, nuestra propia autoridad de certificación, de esta forma no necesitamos ninguna otra que nos respalde y para realizar las pruebas de rendimiento con este tipo de certificados es suficiente para estudiar la eficiencia.

Para generar los certificados RSA y ECDSA usaremos las herramientas que OpenSSL nos proporciona para ello. En la Tabla 5.2 podemos ver los comandos que usaremos para realizar los diferentes pasos hasta generar los certificados [20].

Comando OpenSSL	Función
genrsa	Generación de la clave privada RSA.
out filename	Nombre del archivo de salida, incluyendo la extensión.
numbits	Tamaño de la clave privada a generar en bits. Se pone como la última opción. Por defecto 512 bits.
ecparam	Manipulación y generación de parámetros ECC.
list_curves	Imprime en pantalla la lista de todos los nombres de los EC parámetros actualmente implementados.
genkey	Genera una clave EC privada usando los parámetros especificados.
name arg	Nombre de la curva a usar.
out *filename*	Especifica el nombre de archivo de salida.
req	Petición de firma de certificado X.509.
new	Genera una nueva petición de certificado.
key *filename*	Especifica el archivo de donde leer la clave privada.
x509	Genera un certificado autofirmado.
days *n*	Especifica el número de días de validez del certificado.
in *filename*	Especifica el archivo de entrada a leer.

Tabla 5.2. Comandos OpenSSl usados

- Los pasos que hemos seguido para crear los certificados **RSA** son:

1. Generar la clave RSA privada con el tamaño de clave que queramos.

   $>> openssl genrsa -out *filename numbits*

   Ejemplo*: $>> openssl genrsa -out key.pem 2048*

2. Realizar una petición para la firma con clave privada especificada.

   $>> openssl req -new -key *filename* –out *filename*

   Ejemplo*: $>> req -new -key key.pem -out csr.pem*

3. Generar certificado autofirmado.

   $>> openssl req -x509 -days 365 -key *filename* -in *filename* –out *filename*

   Ejemplo: *$>> req -x509 -days 365 -key key.pem -in csr.pem -out* certificateRSA2048.pem

Para generar los certificados ECDSA primero tenemos que elegir que curvas elegir para implementarlos.

Las curvas disponibles en Openssl las obtenemos ejecutando en el terminal:

$>> openssl ecparam -list_curves

Obteniendo la lista de curvas disponibles en la versión de Openssl que tenemos instalada:

```
secp112r1 : SECG/WTLS curveovera 112 bit primefield
  secp112r2 : SECG curveovera 112 bit primefield
  secp128r1 : SECG curveovera 128 bit primefield
  secp128r2 : SECG curveovera 128 bit primefield
  secp160k1 : SECG curveovera 160 bit primefield
  secp160r1 : SECG curveovera 160 bit primefield
  secp160r2 : SECG/WTLS curveovera 160 bit primefield
  secp192k1 : SECG curveovera 192 bit primefield
  secp224k1 : SECG curveovera 224 bit primefield
  secp224r1 : NIST/SECG curveovera 224 bit primefield
  secp256k1 : SECG curveovera 256 bit primefield
```

secp384r1 : NIST/SECG curveovera 384 bit primefield
secp521r1 : NIST/SECG curveovera 512 bit primefield
prime192v1: NIST/X9.62/SECG curveovera 192 bit primefield
prime192v2: X9.62 curveovera 192 bit primefield
prime192v3: X9.62 curveovera 192 bit primefield
prime239v1: X9.62 curveovera 239 bit primefield
prime239v2: X9.62 curveovera 239 bit primefield
prime239v3: X9.62 curveovera 239 bit primefield
prime256v1: X9.62/SECG curveovera 256 bit primefield
sect113r1 : SECG curveovera 113 bitbinaryfield
sect113r2 : SECG curveovera 113 bitbinaryfield
sect131r1 : SECG/WTLS curveovera 131 bitbinaryfield
sect131r2 : SECG curveovera 131 bitbinaryfield
sect163k1 : NIST/SECG/WTLS curveovera 163 bitbinaryfield
sect163r1 : SECG curveovera 163 bitbinaryfield
sect163r2 : NIST/SECG curveovera 163 bitbinaryfield
sect193r1 : SECG curveovera 193 bitbinaryfield
sect193r2 : SECG curveovera 193 bitbinaryfield
sect233k1 : NIST/SECG/WTLS curveovera 233 bitbinaryfield
sect233r1 : NIST/SECG/WTLS curveovera 233 bitbinaryfield
sect239k1 : SECG curveovera 239 bitbinaryfield
sect283k1 : NIST/SECG curveovera 283 bitbinaryfield
sect283r1 : NIST/SECG curveovera 283 bitbinaryfield
sect409k1 : NIST/SECG curveovera 409 bitbinaryfield
sect409r1 : NIST/SECG curveovera 409 bitbinaryfield
sect571k1 : NIST/SECG curveovera 571 bitbinaryfield
sect571r1 : NIST/SECG curveovera 571 bitbinaryfield
c2pnb163v1: X9.62 curveovera 163 bitbinaryfield
c2pnb163v2: X9.62 curveovera 163 bitbinaryfield
c2pnb163v3: X9.62 curveovera 163 bitbinaryfield
c2pnb176v1: X9.62 curveovera 176 bitbinaryfield
c2tnb191v1: X9.62 curveovera 191 bitbinaryfield
c2tnb191v2: X9.62 curveovera 191 bitbinaryfield
c2tnb191v3: X9.62 curveovera 191 bitbinaryfield
c2pnb208w1: X9.62 curveovera 208 bitbinaryfield
c2tnb239v1: X9.62 curveovera 239 bitbinaryfield
c2tnb239v2: X9.62 curveovera 239 bitbinaryfield
c2tnb239v3: X9.62 curveovera 239 bitbinaryfield
c2pnb272w1: X9.62 curveovera 272 bitbinaryfield
c2pnb304w1: X9.62 curveovera 304 bitbinaryfield
c2tnb359v1: X9.62 curveovera 359 bitbinaryfield
c2pnb368w1: X9.62 curveovera 368 bitbinaryfield
c2tnb431r1: X9.62 curveovera 431 bitbinaryfield
wap-wsg-idm-ecid-wtls1: WTLS curveovera 113 bitbinaryfield
wap-wsg-idm-ecid-wtls3: NIST/SECG/WTLS curveovera 163 bitbinaryfield
wap-wsg-idm-ecid-wtls4: SECG curveovera 113 bitbinaryfield
wap-wsg-idm-ecid-wtls5: X9.62 curveovera 163 bitbinaryfield
wap-wsg-idm-ecid-wtls6: SECG/WTLS curveovera 112 bit primefield
wap-wsg-idm-ecid-wtls7: SECG/WTLS curveovera 160 bit primefield
wap-wsg-idm-ecid-wtls8: WTLS curveovera 112 bit primefield
wap-wsg-idm-ecid-wtls9: WTLS curveovera 160 bit primefield
wap-wsg-idm-ecid-wtls10: NIST/SECG/WTLS curveovera 233 bitbinaryfield
wap-wsg-idm-ecid-wtls11: NIST/SECG/WTLS curveovera 233 bitbinaryfield
wap-wsg-idm-ecid-wtls12: WTLScurvsovera 224 bit primefield
Oakley-EC2N-3:
IPSec/IKE/Oakleycurve #3overa 155 bitbinaryfield.
NotsuitableforECDSA.
Questionableextensionfield!
Oakley-EC2N-4:
IPSec/IKE/Oakleycurve #4overa 185 bitbinaryfield.
NotsuitableforECDSA.
Questionableextensionfiel

De esta lista de curvas elegimos las que correspondan con los valores de las claves que queremos usar. En nuestro caso las curvas resaltadas son las que hemos usado pues son las que corresponden con los valores de bits de clave que queremos comparar.

- Los pasos usados para crear los certificados **ECDSA** (ECC) son:

  1. Generar la clave EDSA privada con el tamaño de clave que queramos mediante el uso de su curva correspondiente.

     $>> openssl ecparam -genkey -name prime256v1 -out key.pem

     Ejemplo: $>> openssl ecparam -genkey -name secp256k1 -out key.pem

  2. Realizar una petición para la firma con clave privada especificada.

     $>> openssl req -new -key *filename* –out *filename*

     Ejemplo: $>> openssl req -new -key key.pem -out csr.pem

  3. Generar certificado autofirmado.

     $>> openssl req -x509 -days *n* -key *filename* -in *filename* –out *filename*

     Ejemplo: $>> openssl req -x509 -days 365 -key key.pem -in csr.pem -out certificateECC256.pem

Los certificados que hemos creado, en base a llevar a cabo la comparativa reflejada en la Tabla 5.1, para realizar las pruebas los tenemos en la Tabla 5.3.

Certificados RSA	Certificados ECC
certificateRSA1024.pem	certificateECC160.pem
certificateRSA2048.pem	certificateECC224.pem
certificateRSA3072.pem	certificateECC256.pem
certificateRSA7680.pem	certificateECC384.pem
certificateRSA15360.pem	certificateECC512.pem

Tabla 5.3. Certificados RSA y ECC creados.

Viendo el nombre del certificado podemos saber el tipo de certificado que es y el tamaño de clave que implementa.

Una vez que tenemos los certificados creados debemos ponerlos junto con las clave (archivos keyRSA.pem o keyRSA.pem) en las carpetas donde Apache guarda las claves de los certificados y a su vez decirle al servidor la ruta donde encontrar dicho certificados. Esto se realiza en el archivo de configuración *default-ssl.conf*. Durante las pruebas se va cambiando la ruta de las llaves y certificados en dicho archivo para que Apache use el certificado que queremos probar en cada momento.

A continuación podemos ver la parte modificada del archivo de configuración de Apache *default-ssl.conf* para el par de certificados *certificateRSA1024.pem-certificateECC160.pem.*

```
SSLEngine on
SSLProtocol all -SSLv2 -SSLv3
SSLHonorCipherOrder     on
SSLCompression          of

#self-signed certificates created to tests
#ECC certificates

SSLCertificateFile  /etc/ssl/certs/certificateECC160.pem
SSLCertificateKeyFile /etc/ssl/private/keyECC160.pem

#RSA certificates

SSLCertificateFile  /etc/ssl/certs/certificateRSA1024.pem
SSLCertificateKeyFile /etc/ssl/private/keyRSA1024.pem
```

Para el resto de parejas de certificados se realiza de la misma manera, cambiando el nombre de los certificados por los nombres de los que vayamos a probar.

5.1.2 Descripción de la prueba de tiempo

Para realizar las medidas de tiempo de cada handshake hemos programado el cliente Android de tal forma que calcula el tiempo empleado en dicho proceso. Para ello, tal y como comentamos en el capítulo 3 de este proyecto, se establece la hora en milisegundos justo antes y en el momento que finaliza el handshake, la diferencia de ambos valores es el tiempo que tarda en realizarse.

Las pruebas se realizan mediante la conexión a la misma red WLAN a la que está conectado el servidor Apache, por lo que cada medida la realizaremos diez veces y promediaremos para minimizar el efecto del posible retardo introducido por la red.

Usaremos todos los cifradores soportados por Android, por lo que no estableceremos ningún cifrador concreto en el servidor Apache, siendo este el que elija el más adecuado para la conexión según el certificado y la clave a usar para realizar cada *handshake.*

Instalamos el cliente Android que hemos desarrollado en el dispositivo Android que vamos a usar para las pruebas. En nuestro se usó un dispositivo *bq* sobre con el sistema operativo Android versión 4.4.2 o también denominada Android Kit-Kat. Esta versión es actualmente la más usada.

Una vez instalada, la abrimos y nos aparecerá la interfaz principal tal y como podemos ver en la Figura 5.1.

Figura 5.1 Interfaz del cliente Android programado

En el primer campo pondremos la url de nuestro servidor, y en el segundo el nombre de uno de los certificados de la Tabla 5.3. De esta forma, la aplicación sabe que certificado usar para realizar las conexiones. El tercer campo lo dejaremos en blanco pues será usado en la tercera parte de las pruebas.

Una vez que tenemos elegido el certificado que queremos usar en el archivo de configuración TLS/SSL de apache, reiniciamos el servidor, tecleamos la url del servidor, ponemos el nombre del archivo del certificado a usar y pulsamos conectar. Tras esto el cliente Android realiza la conexión usando dicho certificado diez veces escribiendo el resultado de la prueba en un archivo *.txt* localizado en la ruta `/data/data/android.navegador/files/resultTime.txt` del sistema de archivos del dispositivo móvil. Dicho archivo presenta la siguiente estructura:

```
CERTIFICADO:certificateECC160.pem
- Cipher:  TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- Tiempo handshake:  75 (ms)
----------
CERTIFICADO:certificateECC160.pem
- Cipher:  TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- Tiempo handshake:  64 (ms)
----------
CERTIFICADO:certificateECC160.pem
- Cipher:  TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- Tiempo handshake:  64 (ms)
----------
```

Vemos que recoge el tiempo empleado en realizar *handshake* por cada certificado y el cifrador usado en la conexión.

En este archivo *.txt* se recoge uno a continuación de otro las 10 repeticiones con cada certificado de tal forma que en un único archivo tenemos todas las medidas de tiempo.

5.1.3 Medidas de tiempo

Tras ejecutar el navegador Android programado tal y como hemos comentado en el apartado anterior hemos obtenido el archivo *.txt* con todos los resultados. Se han analizado estos resultados calculando el valor medio de cada una de las 10 repeticiones obteniendo los valores que veremos a continuación. Para comprobar las medidas de tiempo completas podemos consultar el Apéndice de este proyecto.

- Valores medios de tiempo obtenidos para los certificados RSA:

Bits de clave RSA	Valor medio de tiempo en realizar el handshake (en milisegundos)
1024	36,6
2048	45,3
3072	52,8
7680	201,1
15360	1047,4

Tabla 5.4. Valor medio de tiempos RSA

- Valores medios de tiempo obtenidos para los certificados ECDSA:

Bits de clave ECDSA	Valor medio de tiempo en realizar el handshake (en milisegundos)
160	40,1
224	46,9
256	49,7
384	65
512	96,2

Tabla 5.5 Valor medio tiempos ECDSA

Vemos que en ambos casos los valores de tiempo empleado en hacer el handshake aumentan conforme la clave del certificado se vuelve más compleja cosa totalmente lógica pues al aumentar el número de bits de clave aumentamos las complejidad de la misma y por tanto es necesario más tiempo para realizar el descifrado.

Tal y como vimos en apartado anteriores, existe una equivalencia entre la seguridad que ofrecen los certificados RSA y los ECDSA, y es esta equivalencia la que usaremos para estudiar la diferencia entre ambos tipos de certificados. Recordemos dicha equivalencia en seguridad de la clave en la Tabla 5.1.

Comparamos los valores de ambos tipos de certificados para los mismos niveles de seguridad:

Bits de claves	Tiempo en realizar handshake (en milisegundos)
RSA 1024	36,6
ECC 160	40,1
RSA2048	45,3
ECC224	46,9
RSA3072	52,8
ECC256	49,7
RSA7680	201,1
ECC384	65
RSA15360	1047,4
ECC512	96,2

Tabla 5.6. Comparación de resultados RSA/EDCSA

Para los valores más bajos de ambos tipos de certificados vemos que la diferencia de tiempos entre el uso de ambos tipos de certificados es muy baja siendo prácticamente iguales.

Para los valores más altos si encontramos grandes diferencias de tiempo entre el uso de ECC con respecto a RSA para niveles de seguridad similares. Podemos ver esto más claro en la Figura 5.2:

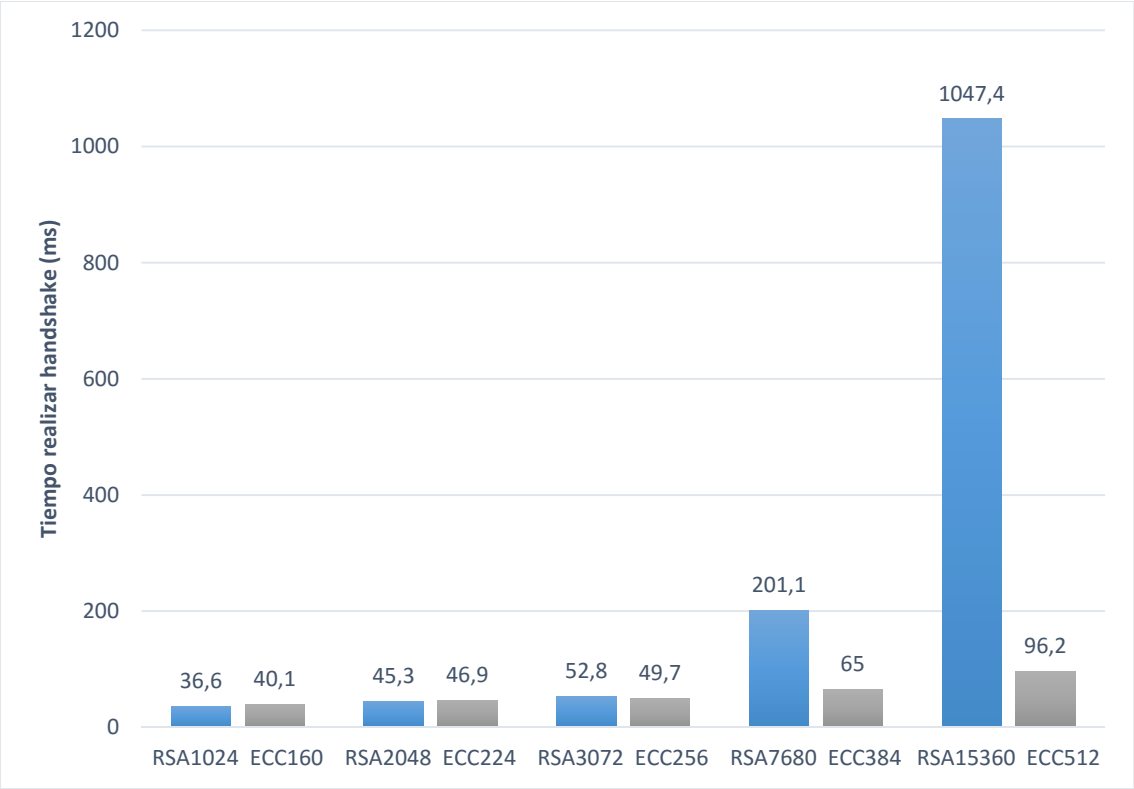


Figura 5.2. Comparativa de tiempos handshak TLS entre RSA y ECDSA

Aquí vemos claramente como en cuanto aumentamos el nivel de seguridad RSA se dispara el tiempo de ejecución en realizar el handshake en las conexiones TLS, por lo que en estos niveles ECDSA (ECC) ofrece una mayor eficiencia pues requiere mucho menos tiempo para realizar la conexión.

Para valores bajos de vemos que prácticamente nos lleva el mismo tiempo realizar conexiones mediante RSA que ECDSA, sin embargo, como ya sabemos, el cifrado mediante ECC es más difícil de romper que el usado por RSA por lo que a igualdad de tiempos y siendo más seguro ECDSA, podemos decir que para valores bajos de bits de clave también es más eficiente ECC, aunque todavía tenemos que analizar las pruebas de energía para confirmar esto.

5.1.4 Descripción de la prueba de energía

De todos los valores que PowerTutor nos proporciona, nos interesa el valor de la energía empleada por nuestra aplicación.

Al no poder separar diferentes procesos en nuestro navegador y obtener los valores de energía para el proceso de *handshake* solamente, tendremos que anotar el valor de energía para cada certificado de forma individual e ir iniciando la aplicación *PowerTutor* manualmente para cada uno y comparar los resultados. Cómo el único

proceso que cambia de usar un certificado u otro es el "handshake" podremos comparar los diferentes valores de forma sencilla.

Para asegurarnos una mayor exactitud en la medida de energía, haremos diez intentos para cada certificado, calculando el valor medio y la desviación típica. Con estos valores finales podremos compararlos entre ellos y así establecer el uso de energía de cada tipo de certificado.

Tras realizar la prueba de la forma que hemos explicado obtenemos los resultados del consumo de energía de la aplicación para cada uno de los certificados que hemos creado. Hemos realizado diez veces cada medida para minimizar el efecto que pudieran provocar los retardos de la WLAN y las inexactitudes de realizar las medidas mediante una aplicación externa y sobre todo el funcionamiento de nuestro navegador.

Por ello para realizar estas pruebas hemos modificado el navegador de tal forma que no escriba ningún archivo de texto pues la información que queremos nos la dará *PowerTutor*. De esta forma el navegador sólo se encarga de realizar el handshake con el servidor Apache. A pesar de eso las medidas de energía no son exactamente del momento de handshake, aun así, con los valores que obtendremos esperamos sacar alguna conclusión sobre el consumo de energía de los certificados que queremos probar.

5.1.5 Medidas de energía

El valor medio de energía obtenido en las pruebas lo podemos ver a continuación. El valor completo de todas las repeticiones realizadas lo podemos encontrar en el apéndice b de este proyecto.

- Valores medios de energía consumida obtenidos para los certificados RSA:

Bits de clave RSA	Valor medio de energía consumida en el handshake (en milijulios)
1024	27,9
2048	34,9
3072	46,6
7680	61,3
15360	113,6

Tabla 5.7. Valor medio de energía usando RSA

- Valores medios de energía consumida obtenidos para los certificados ECDSA:

Bits de clave ECDSA	Valor medio de energía consumida en realizar el handshake (en milijulios)
160	43,8
224	51,4
256	47,2
384	68,9
512	52

Tabla 5.8. Valor medio de energía usando ECDSA

Al igual que hemos realizado en las medidas de los tiempos empleados en las conexiones comparamos los valores de energía consumida entre los certificados que ofrecen un nivel de seguridad similar:

Bits de claves	Energía consumida handshake (en milijulios)
RSA 1024	27,9
ECC 160	43,8
RSA2048	34,9
ECC224	51,4
RSA3072	46,6
ECC256	47,2
RSA7680	61,3
ECC384	68,9
RSA15360	113,6
ECC512	52

Tabla 5.9. Comparación de resultados RSA/EDCSA

Vemos esto de forma más clara en la Figura 5.3.

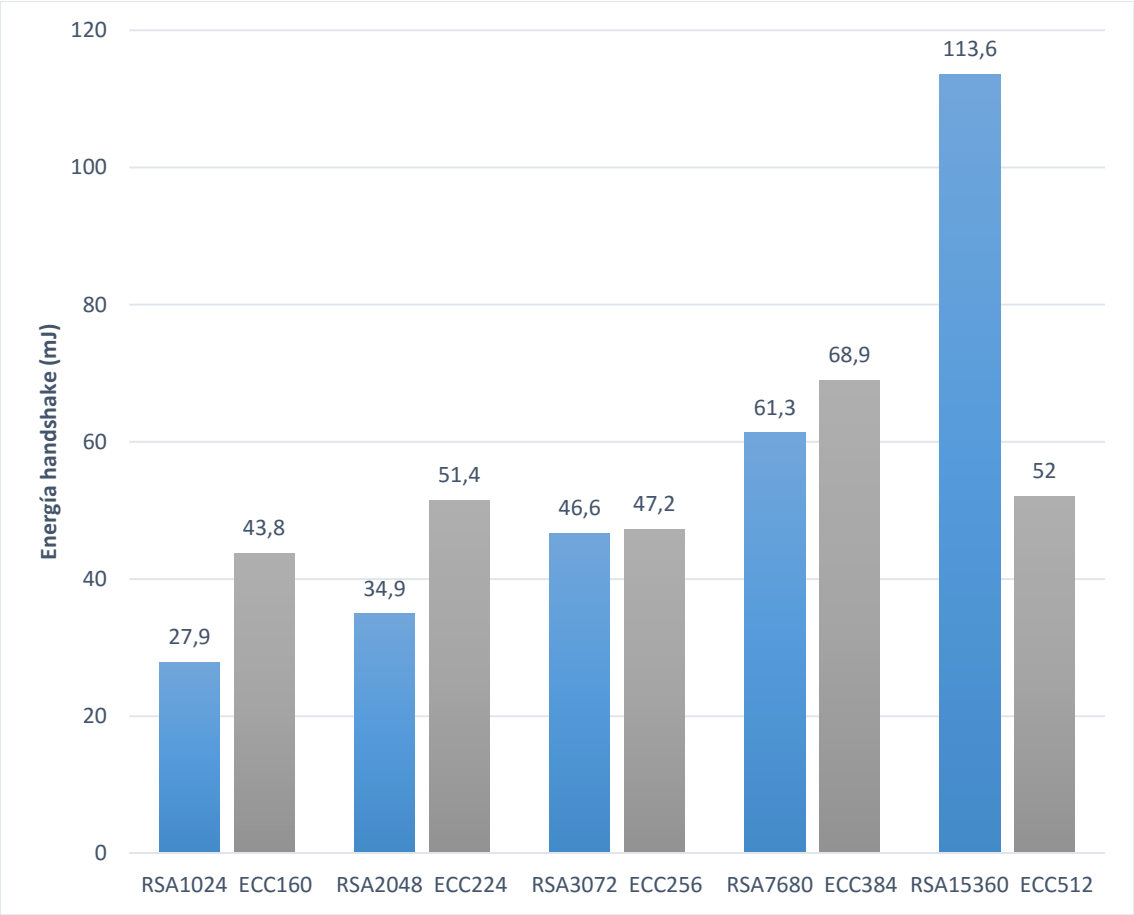


Figura 5.3 Comparativa de energía consumida entre RSA y ECDSA

A la vista de los resultados obtenidos obtenemos un resultado parecido al obtenido en el apartado de las medidas de tiempos. Para valores bajos de bits de clave los valores energía consumida son muy parecidos e incluso para ECC algo superiores. La diferencia radica en los valores altos y por tanto donde el cifrado es mayor. Ahí si vemos como RSA tiene un gasto energético muy superior a los certificados ECDSA con un nivel de seguridad parecido. Esto es importante, sobre todo en Android, pues si queremos usar un nivel de seguridad muy alto y usamos RSA el gasto energético se dispara, por lo que en dispositivos móviles que dependen de una batería para funcionar es algo fundamental a tener en cuenta. Por ello, para estos niveles de seguridad sería muy recomendable el uso de ECC por encima de RSA.

En el siguiente apartado veremos con más detalles la diferencias entre el uso de los dos tipos de certificados a la luz de las pruebas de tiempo y energía que hemos realizado.

5.1.6 Discusión de resultados

Tal y como comentamos en los primeros capítulos de este proyecto la principal diferencia entre el uso de RSA y ECC es que para un mismo nivel de seguridad, los certificados ECDSA requieren una clave de menos bits (muchos menos según vamos aumentando la seguridad y por tanto el número de bits) que los certificados RSA, por tanto en un principio son más eficaces que los RSA.

Con los resultados obtenidos en las pruebas vemos que además de la mayor seguridad que pueden aportar con un coste criptográfico menor, pueden ser más eficientes en términos de tiempo y energía, algo fundamental en los dispositivos móviles donde el bajo consumo energético y la rapidez son algo fundamental.

Donde mayor vemos esta ventaja es en las medidas de tiempo donde según vamos aumentando la seguridad, el tiempo que tarda en realizar el handshake con un certificado ECDSA es mucho menos que el empleado con un certificado RSA. Esto es algo muy importante porque influye en la velocidad de carga de contenido de la red en el dispositivo Android, algo fundamental para el buen rendimiento de este tipo de conexiones en dispositivos móviles.

En los valores de energía vemos mayor igualdad en los valores medios y bajos de seguridad seguramente debido al tiempo extra que introduce el uso manual de Powertutor, sin embargo, el consumo de energía se dispara en el caso de RSA para los niveles altos producido por el alto coste criptográfico que requiere una clave con ese número de bits y por tanto un alto nivel de energía requiere el dispositivo Android para poder realizar dicha operación. En los niveles medios y bajos vemos mayor igualdad, e incluso un poco más alto el consumo de energía de parte de los certificados ECDSA. En los niveles bajos esto podemos justificarlos debido a que los valores de bits entre unos y otros no difieren en demasía, aun así la ventaja de ECC es mayor pues requiere un nivel de energía más o menos igual para valores de clave menores, con lo que comprobamos la eficiencia criptográfica de este tipo de certificados.

En definitiva, podemos decir que el uso de certificados ECDSA (ECC) es más eficiente que el uso de RSA ya que presenta las siguientes ventajas:

- Mismo o menor consumo energético para ofrecer la misma seguridad pero con un coste criptográfico menor (necesidad de menor número de bits de clave)
- Menos consumo de tiempo manteniéndose más o menos estable conforme se aumenta el número de bits de clave.

Por tanto, y a la vista de los resultados anteriores, podemos decir que su uso es ventajoso en dispositivos Android con respecto al uso de RSA.

5.2 Medida de eficiencia en conexiones con servidores comerciales

En este apartado estudiaremos el soporte que tienen los servidores comerciales más visitados con respecto al uso de certificados ECC usando para realizar las conexiones y establecer los parámetros deseado el navegador que hemos programado.

5.2.1 Descripción de la prueba

Obtenemos la lista de servidores más populares visitando la web:

http://www.alexa.com/topsites

De la lista que aparece se han seleccionado los 10 primeros que serán los que usaremos para comprobar el soporte a ECC. Los servidores comerciales elegidos, por orden de mayor a menos visitas, son:

1 Google.com
2 Facebook.com
3 Youtube.com
4 Baidu.com
5 Yahoo.com
6 Amazon.com
7 Wikipedia.org
8 Qq.com
9 Taobao.com
10 Twitter.com

Para poder probar el soporte de estos servidores comerciales al uso de ECC debemos forzar a que nos envíen el certificado en un formato específico, en nuestro caso ECDSA. Para ello, anunciaremos que el cliente Android sólo soporta un *ciphersuite* específico, que en nuestro caso será un *ciphersuite* que necesite cifrado ECC. Para ello, usaremos los cifradores ECDSA que soporta el API de Android (Tabla 5.10).

Para comenzar la prueba iniciamos el cliente Android apareciéndonos la misma pantalla de inicio que las anteriores pruebas. En este caso, en url escribimos la dirección del servidor comercial que queremos probar, el segundo campo lo dejamos vacío, de esta forma el cliente Android detectará que queremos realizar una conexión a un servidor comercial y establecerá la suite de cifradores a la lista de cifradores que soportan ECC.

Al realizar la conexión con un servidor externo comercial, el cliente Android se encargará de ir probando la conexión con cada uno de los cifradores dados y volcará el

resultado de dichas conexiones en un archivo .txt que se guarda en */data/data/com.navegador/files* dentro del dispositivo Android. Dicho archivo de texto presenta la siguiente forma:

```
Servidor web:google.com
- Cipher:  TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
- Conexión:  Error
---------------------------
Servidor web:google.com
- Cipher:  TLS_ECDH_ECDSA_WITH_NULL_SHA
- Conexión:  Connect OK
---------------------------
Servidor web:google.com
- Cipher:  TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- Conexión:  Connect OK
---------------------------
Servidor web:facebook.com
- Cipher:  TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- Conexión:  Error
---------------------------
```

Si observamos el archivo, comprobamos que en aquellos donde en el campo de conexión aparezca un "OK" significa que ese cifrador es admitido por el servidor comercial probado. De ahí obtendremos los cifradores ECC soportados por cada servidor comercial.

5.2.2 Soporte cifradores ECDSA

Tras realizar la prueba como hemos comentado en apartado anterior obtenemos los cifradores que soportan cada uno de los servidores probados. En la Tabla 5.10 vemos resaltados los cifradores ECDSA soportados, donde comprobamos que son los mismos para los diez servidores probados.

Cifradores Android ECDSA
TLS_ECDH_ECDSA_WITH_RC4_128_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_NULL_SHA
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_NULL_SHA

Tabla 5.10. Cifradores ECDSA soportados por Android. [18]

5.5.3 Discusión

Tras la realización de la prueba con servidores comerciales podemos discutir diferentes cuestiones a luz de los resultados obtenidos:

- **Soporte ECC en servidores comerciales**

Comprobamos que los servidores comerciales probados poseen un soporte ECC mayor que tres de los cuatros navegadores comerciales probados (Firefox, Chrome y Ópera), por lo que podemos concluir que en estos servidores comerciales poseen un soporte a este tipo de conexiones más que suficiente para poder ser usada desde los navegadores y aplicaciones Android, incluso podemos decir que si los navegadores comerciales de Android decidieran aumentar su soporte ECC estas webs estarían preparadas para ello.

- **Cumplimiento recomendaciones NIST**

NIST (*National Institute of Standards and Technology*)[21] establece recomendaciones sobre el uso de los algoritmos y protocolos de seguridad. Para ello se basa en la posibilidad de que los problemas matemáticos en los que están basados vayan siendo resueltos de forma relativamente sencilla.

Las recomendaciones de NIST para los algoritmos y protocolos de seguridad en las conexiones TLS las podemos ver en la Tabla 5.11.

Claves (bits)	Algoritmo	Uso restringido	Nuevo estado
80	3DES-2 llaves	2011-2014	2015 NO usar
112	3DES-3 llaves	ninguno	aceptable
112	AES-128	ninguno	aceptable
mayor a 112	AES-192	ninguno	aceptable
mayor a 112	AES-256	ninguno	aceptable
80	RSA (entre 1024 y 2048)	2011-2013	2013 NO usar
80	ECDSA (entre 160 y 224)	2011-2013	2013 NO usar
mayor a 112	RSA (2048 o mayor)	ninguno	aceptable
mayor a 112	ECDSA (mayor a 224)	ninguno	aceptable

Tabla 5.11. Recomendaciones NIST 2012 [21].

Tras observar las recomendaciones podemos concluir que cifradores ECDSA de Android las cumplen y cuales no. En la Tabla 5.12 tenemos la lista completa de dichos cifradores.

[21] http://www.nist.gov/

Cifradores ECDSA Android	Sigue recomendación NIST
TLS_ECDH_ECDSA_WITH_RC4_128_SHA	No
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDHE_ECDSA_WITH_NULL_SHA	No
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	No
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	Si se usa clave mayor de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDH_ECDSA_WITH_NULL_SHA	No
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	Si se usa clave de 112 bits y clave ECDSA mayor de 224 bits

Tabla 5.12. Comparación cifradores ECDSA soportados por servidores comerciales con respecto a las recomendaciones NIST

Comprobamos como la mayoría de los cifradores soportados por los servidores comerciales (cifradores resaltados) probados siguen las recomendaciones de NIST, aunque cuatro de ellos ya están obsoletos según estas recomendaciones y por tanto, pueden ser inseguros.

Con respecto al resto de cifradores que implementa ECDSA de Android vemos que si cumplen las recomendaciones de NIST y que por tanto, serán seguros durante años según sus previsiones, siempre y cuando usen una clave ECDSA de 224 bits o mayor. Esto demuestra que Android está preparado ante la posibilidad de que los cifradores vayan siendo inseguros.

- **Soporte TLS**

Si observamos la lista de cifradores con claves ECDSA soportados por Android (Tabla 5.10.) que hemos usado para probar el soporte de los servidores comerciales a ECC vemos que todos requieren el uso de TLS. Esto es debido a que ECDSA no permite el uso de SSL para realizar el intercambio de claves como podemos ver en la Tabla 5.13.

Algoritmo	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2
DHE-RSA					
RSA	Sí	Sí	Sí	Sí	Sí
DH-RSA	No	Sí	Sí	Sí	Sí
ECDHE-RSA					
ECDH-RSA	No	No	Sí	Sí	Sí
DHE-DSS					
DH-DSS	No	Sí	Sí	Sí	Sí
ECDHE-ECDSA					
ECDH-ECDSA	No	No	Sí	Sí	Sí

Tabla 5.13. Autenticación e intercambio de claves. [12]

Por ello, vemos que la implementación que realiza Android de ECC requiere el uso de TLS.

Teniendo en cuenta esto, podemos concluir que Android ofrece un soporte de seguridad alto y acorde con las recomendaciones y estándares.

Capítulo 6

PLANIFICACIÓN Y PRESUPUESTO

Este capítulo detallaremos, por un lado a la planificación y las fases de desarrollo de la realización de este proyecto, y por otro, desglosaremos el presupuesto detallando los costes de personal, del material y costes totales.

6.1 Planificación

A continuación, se detalla la planificación seguida para la realización de este proyecto.

El proyecto se ha desarrollado en cuatro fases principales:

- **Fase 1: Estudio y documentación**

  En esta primera fase se realiza el estudio previo sobre qué se quiere hacer y a partir de ahí realizar el plan de trabajo. Finalmente se procede al proceso de documentación. Consta de las siguientes subfases:

  - Estudio previo
  - Desarrollar plan de trabajo
  - Documentación

- **Fase 2: Análisis y diseño**

  En la segunda fase se realiza el análisis de todas las tecnologías necesarias en el proyecto y el diseño de las herramientas necesarias para realizar las pruebas. Consta de las siguientes subfases:

  - Estudio y diseño servidor Apache y OpenSSL
  - Estudio y diseño scripts Apache
  - Estudio API Android y plataforma Android Studio
  - Diseño navegador en Android
  - Estudio navegadores comerciales Android
  - Diseño entorno de pruebas

- **Fase 3: Implementación y pruebas**

  En la tercera fase se realizan las pruebas y se sacan las conclusiones pertinentes. Consta de las siguientes subfases:

  - Implementación entorno de pruebas
  - Recogida de datos de la prueba con los navegadores comerciales
  - Recogida de datos de la prueba con el navegador programático
  - Organización datos obtenido en las pruebas
  - Comparativa de datos y conclusiones

- **Fase 4: Realización de la memoria**

  En la cuarta y última fase se redacta la memoria del proyecto. Consta de las siguientes subfases:

  - Redacción de la memoria
  - Revisión y corrección de la memoria.

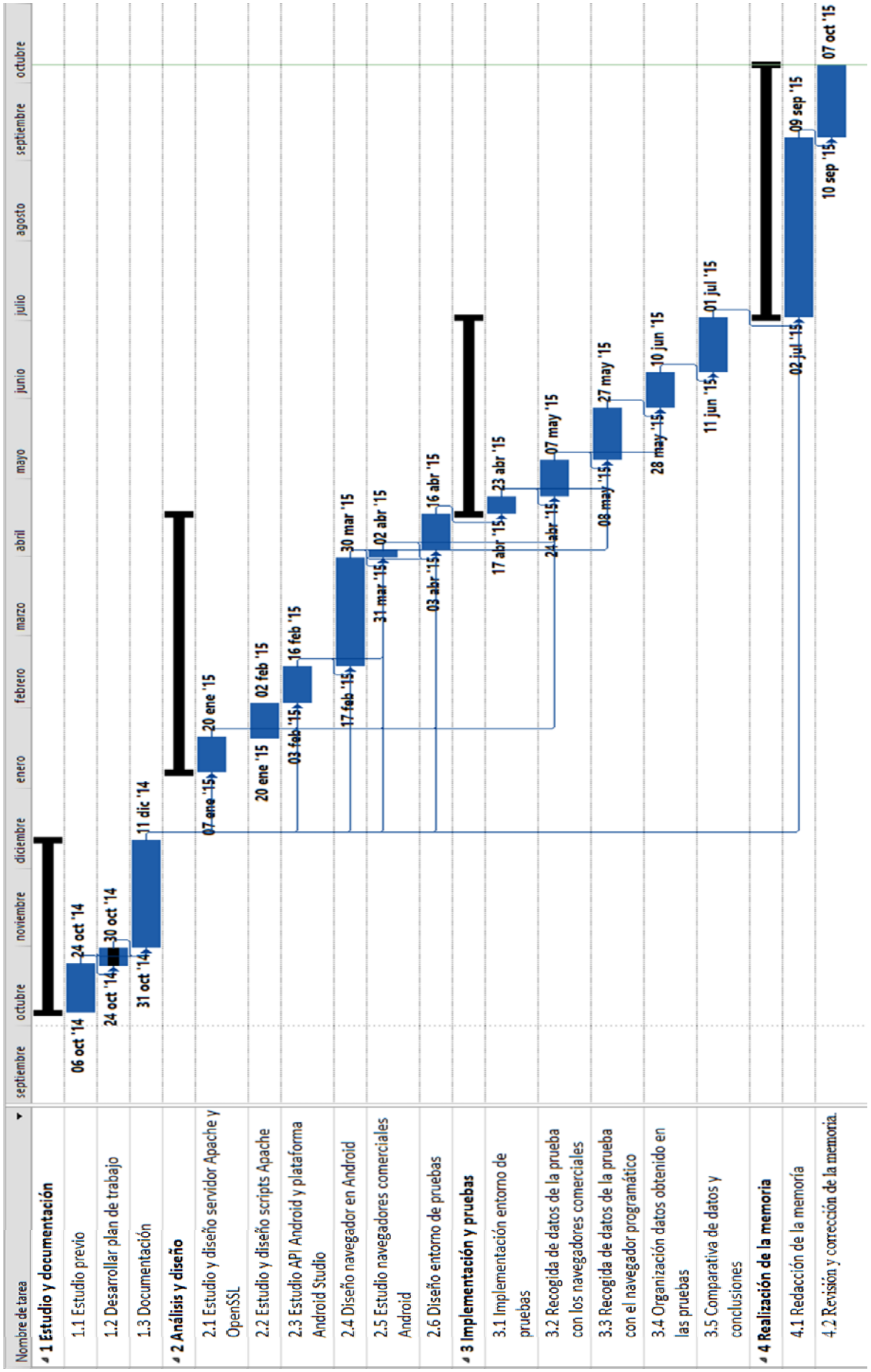
En la Tabla 7.1 podemos ver la división completa en tareas y subtareas realizada a la hora de planificar este proyecto, así como la duración en días de cada una de ellas.

Nombre de tarea	Duración
1 Estudio y documentación	**49 días**
1.1 Estudio previo	15 días
1.2 Desarrollar plan de trabajo	5 días
1.3 Documentación	30 días
2 Análisis y diseño	**72 días**
2.1 Estudio y diseño servidor Apache y OpenSSL	10 días
2.2 Estudio y diseño scripts Apache	10 días
2.3 Estudio API Android y plataforma Android Studio	10 días
2.4 Diseño navegador en Android	30 días
2.5 Estudio navegadores comerciales Android	3 días
2.5 Diseño entorno de pruebas	10 días
3 Implementación y pruebas	**54 días**
3.1 Implementación entorno de pruebas	5 días
3.2 Recogida de datos de la prueba con los navegadores comerciales	10 días
3.3 Recogida de datos de la prueba con el navegador programático	10 días
3.4 Organización datos obtenido en las pruebas	10 días
3.5 Comparativa de datos y conclusiones	15 días

Nombre de tarea	Duración
4 Realización de la memoria	**70 días**
4.1 Redacción de la memoria	50 días
4.2 Revisión y corrección de la memoria.	20 días

Tabla 6.1. División tareas y subtareas del proyecto.

A continuación tenemos la planificación de las diferentes tareas y subtareas y las dependencias entre ellas en el siguiente diagrama de Gantt:


Nombre de tarea
septiembre
octubre
noviembre
diciembre
enero
febrero
marzo
abril
mayo
junio
julio
agosto
septiembre
octubre
1 Estudio y documentación
1.1 Estudio previo
1.2 Desarrollar plan de trabajo
1.3 Documentación
2 Análisis y diseño
2.1 Estudio y diseño servidor Apache y OpenSSL
2.2 Estudio y diseño scripts Apache
2.3 Estudio API Android y plataforma Android Studio
2.4 Diseño navegador en Android
2.5 Estudio navegadores comerciales Android
2.6 Diseño entorno de pruebas
3 Implementación y pruebas
3.1 Implementación entorno de pruebas
3.2 Recogida de datos de la prueba con los navegadores comerciales
3.3 Recogida de datos de la prueba con el navegador programático
3.4 Organización datos obtenido en las pruebas
3.5 Comparativa de datos y conclusiones
4 Realización de la memoria
4.1 Redacción de la memoria
4.2 Revisión y corrección de la memoria.
06 oct '14
24 oct '14
24 oct '14
30 oct '14
31 oct '14
11 dic '14
07 ene '15
20 ene '15
20 ene '15
02 feb '15
03 feb '15
16 feb '15
17 feb '15
30 mar '15
31 mar '15
02 abr '15
03 abr '15
16 abr '15
17 abr '15
23 abr '15
24 abr '15
07 may '15
08 may '15
27 may '15
28 may '15
10 jun '15
11 jun '15
01 jul '15
02 jul '15
09 sep '15
10 sep '15
07 oct '15

La duración total de este proyecto ha sido de 12 meses, 52 semanas. Considerando que durante cada semana se han trabajado 5 días, y que cada día, se han dedicado 5 horas al proyecto, el número de horas necesarias para el desarrollo del proyecto han sido 1300 horas

6.2 Presupuesto

A continuación se incluye la hoja de presupuesto del proyecto con los costes del personal, los costes materiales y otros costes directos.

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor: Jesús Manuel Calvo Corrales

2.- Departamento: Ingeniería Telemática

3. – Descripción del Proyecto:

- Título Estudio de rendimiento de conexiones TLS con ECC en dispositivos Android
- Duración (meses) 12

Tasa de costes indirectos: 20%

4. – Presupuesto total del Proyecto (valores en EUROS)

45.457,14 Euros

5. – Desglose presupuestario (costes directos)

PERSONAL						
Apellidos y nombre	N.I.F. (no rellenar - solo a titulo informativo)	Categoría	Dedicación (hombres ,mes) [a)]	Coste hombre mes	Coste (Euro)	Firma de conformidad
Calvo Corrales, Jesús Manuel		Ingeniero Junior	8,8	2.694,39	23.710,63	
Arias Cabarcos, Patricia		Ingeniero Senior	3,25	2.845,82	9.248,92	
Almenares Mendoza, Florina		Ingeniero Senior	3,25	2.845,82	9.248,92	
		Hombres mes	**8,8**	**Total**	**42.208,47**	

[a)] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS					
Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Ordenador portátil	900,00	100	12	60	140,00
Terminal móvil	220,00	100	12	60	44,00
Router	54.54	100	12	60	10,91
				Total	**194,91**

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

OTROS COSTES DIRECTOS DEL PROYECTO[e]		
Descripción	Empresa	Costes imputable
Conexión ADSL	Movistar	533,76
	Total	**533,76**

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,…

6. – Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	42.208,47
Amortización	194,91
Subcontratación de tareas	0
Costes de funcionamiento	533,76
Costes Indirectos	2.520
Total	**45.457,14**

El presupuesto total de este proyecto asciende a la cantidad de **CUARENTA Y CINCO MIL CUATROCIENTOS CINCUENTA Y SIETE CON 14 EUROS.**

Leganés a 28 de Octubre de 2015

El ingeniero proyectista

Fdo. Jesús Manuel Calvo Corrales

Capítulo 7

CONCLUSIONES Y TRABAJO FUTURO

En este capítulo detallaremos las principales conclusiones obtenidas tras la realización de este estudio.

También se darán unas líneas generales desde las que partir para continuar desarrollando este estudio.

7.1 Conclusiones

La seguridad en las conexiones en Internet es algo fundamental y más si estamos hablando de dispositivos móviles. Muchos datos privados dependen de dichas conexiones por lo que puedan ser interceptadas es algo que hay que tener muy en cuenta. Es por ello que los protocolos y algoritmos que se dedican a implementar esta seguridad son fundamentales en cualquier conexión.

Sin embargo, el hecho de que las conexiones sean lo más seguras posibles no debería influir en el rendimiento de dichas conexiones, es decir, no excedan en tiempo y energía ciertos valores que las harían poco eficientes. Por ello son muy interesantes los algoritmos ECC pues aportan un equilibrio entre seguridad y rendimiento.

Durante este trabajo hemos estudiado diferentes aspectos de las conexiones usando algoritmos ECC. Dejando aparte las cuestiones teóricas y de seguridad, las cuales ya están muy demostradas, nos hemos centrado en el estudio del rendimiento de este tipo de conexiones y su implementación. Todo centrado en la plataforma móvil más usada actualmente, Android.

Con este trabajo se ha hecho un estudio de la eficiencia de las conexiones ECC en Android para tratar de averiguar si además de ser más seguras que otras utilizadas anteriormente, también son más eficientes o al menos, su coste temporal y/o energético compensa con la seguridad que aporta. Con ello se ha pretendido certificar que este tipo de conexiones son las ideales para usar por defecto en dispositivos móviles, los cuales por sus propias características necesitan rapidez y poco consumo energético.

Hemos comprobado que la implementación de ECC se ha desarrollado y extendido usándose por igual tanto en los dispositivos móviles como en los servidores de forma que es fácil determinar y monitorizar su uso.

Android implementa los algoritmos ECC de diferentes formas permitiendo su uso tanto en certificado (ECDSA) como en el cifrado del canal de intercambio (ECDH/ECDHE), por lo que presenta un completo desarrollo de los algoritmos ECC.

En las conexiones TSL no sólo es más eficiente y sobre todo más seguro ECC según la teoría, sino que también lo es en la práctica, sobre todo a la hora del cifrado del canal de intercambio ECDHE, donde combinado con las claves y certificados RSA presentan los mejores valores de tiempo al realizar conexiones desde navegadores comerciales.

La mayoría de los navegadores comerciales existentes para Android aprovechan muy bien la capacidad de Android de uso de los algoritmos ECC implementando diferentes cifradores que permiten las conexiones seguras usando esta tecnología.

Se ha demostrado que los certificados ECDSA son un digno heredero de los actualmente más conocidos y usados RSA. Su estudio de forma individual ha demostrado que para un mismo nivel de seguridad presentan un consumo temporal y energético menor, valores que se hacen más patentes de forma que aumentamos los tamaños de clave donde para un mismo nivel de seguridad ECDSA necesita un tamaño

mucho menor y por tanto su coste energético y temporal también lo es. Por tanto, presentan una gran eficiencia al no suponer una gran aumento de tiempo y energía al aumenta la seguridad, sobre todo pensando en un futuro donde será necesario ampliar la seguridad de la claves conforme vayan siendo solucionados los problemas en los que se basan estos algoritmos.

La mayor parte de los cifradores que implementan ECDSA en Android cumplen con las recomendaciones del NIST con respecto a los algoritmos a usar y al tamaño de claves. Además presenta una suite de cifradores preparada para las necesidades de seguridad futuras que actualmente no son necesarias.

También hemos comprobado que los servidores comerciales permiten el uso de los certificados ECDSA.

Por todo ello, podemos concluir que Android hace un uso eficiente de las tecnologías ECC. Aunque en algunos caso todavía es más eficiente el uso de otros algoritmos, en nivel de seguridad que aporta y sobre todo, su poco aumento de valores de tiempo y energía conforme vamos aumentando la seguridad y por tanto, la complejidad de dichos algoritmos, hace que podamos plantear la total migración de los viejos sistemas de cifrado a los actuales basados en ECC en un futuro.

7.2 Trabajo futuro

La primera línea de trabajo futuro que planteamos sería extender este estudio a otras plataformas móviles, tales como IOS, Windows 10, etc. De esta forma se podrían establecer comparaciones entre la eficiencia de las conexiones seguras de los diferentes sistemas operativos para dispositivos móviles y a partir de ahí establecer la eficiencia del uso de ECC en cada uno de ellos. Así podríamos llegar a la conclusión de cuál de ellos hace una implementación más eficiente de ECC.

Otro punto a tratar sería mejorar la forma en que se mide la energía de las conexiones seguras realizando una aplicación para este menester y que midiera el consumo de energía de las conexiones TSL de forma específica. De esta forma se evitaría los errores de medir dichas conexiones, teniendo en cuenta como lo realiza PowerTutor.

Por último, siguiendo las recomendaciones de algunos organismos (como por ejemplo el NIST), se podría estudiar la eficiencia, tanto en Android como en otros sistemas operativos, si la implantación de dichas recomendaciones es eficiente en valores de tiempo y energía de las conexiones seguras.

GLOSARIO

3DES	*Triple Data Encryption Standard*
AES	*Advanced Encryption Standard*
API	*Application Programming Interface*
ASN.1	*Abstract Syntax Notation One*
CA	*Certification Authority*
CGI	*Common Gateway Interface*
CSR	*Certificate Signing Request*
DH	*Diffie–Hellman*
DHE	*Ephemeral Diffie-Hellman*
DSA	*Digital Signature Algorithm*
DSS	*Digital Signature Standard*
ECC	*Elliptic curve cryptography*
ECDH	*Elliptic curve Diffie–Hellman*
ECDHE	*Elliptic Curve Diffie–Hellman Exchange*
ECDSA	*Elliptic Curve Digital Signature Algorithm*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
HTTPD	*Hypertext Transfer Protocol Daemon*
HTTPS	*Hypertext Transfer Protocol Secure*
MAC	*Message authentication code*
MD5	*Message-Digest Algorithm 5*
PHP	*Hypertext Preprocessor*
PKCS	*Public Key Cryptography Standard*
PKI	*Public Key Infrastructure*
RC2	*Rivest Cipher 2*

RFC	*Request for Comments*
RSA	*Rivest Shamir Adleman*
SDK	*Software's Developement Kit*
SHA	*Secure Hash Algorithm*
SRP	*Secure Remote Password*
SSL	*Secure Sockets Layer*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
URL	*Uniform Resource Locator*
WLAN	*Wireless Local Area Network*

REFERENCIAS

[1] De la Fuente, Elma: *Estudio de la eficiencia de protocolos y algortimos de seguridad en Android*. Proyecto Fin de Carrera, Ingeniería Telemática, UC3M, Madrid, Octubre 2015.

[2] Manuel José Lucena López: *Criptografía y Seguridad en Computadores* (Universidad de Jaén, 3ª edición, Junio 2001).

[3] Llorenç Huguet Rotger, Josep Rifà Coma, Juan Gabriel Tena Ayuso: *Criptografía con Curvas elípticas.* https://www.exabyteinformatica.com/uoc/Informatica/Criptografia_avanzada/Criptografia_avanzada_(Modulo_4).pdf [Último acceso: Octubre 2015]

[4] National Institute of Standards and Technology (NIST), *Recommendation for Key Management – Part 1: General (Revision 3)*, NIST 800-57 Table 2, July 2012. http://csrc.nist.gov/publications/nistpubs/800-57/sp80057_part1_rev3_general.pdf [Último acceso: Octubre 2015]

[5] Manuel Báez, Álvaro Borrego, Jorge Cordero, Luis Cruz, Miguel González, Francisco Hernández, David Palomero, José Rodríguez de Llera, Daniel Sanz, Mariam Saucedo, Pilar Torralbo, Álvaro Zapata: *Introducción a Android* (E.M.E. Editorial ©, 2011). Grupo Tecnología UCM.

[6] LEEWei-Meng Lee: *Beginning Android 4 Aplication Develoment* (John Wiley & Sons, Inc., 2012), capítulo 1.

[7] Android Official Web Site. API Android KitKat. Disponible en: https://developer.android.com/about/versions/android-4.4.html. [Último acceso: Octubre 2015]

[8] Android Official Web Site. SDK Android. Disponible en: https://developer.android.com/sdk/index.html [Último acceso: Octubre 2015]

[9] Android Developers Official Web Site: *Security with HTTPS and SSL*. https://developer.android.com/training/articles/security-ssl.html [Último acceso: Octubre 2015]

[10] Chris P.: *Best Android browsers, 2015 edition: speed, features, and design.* http://www.phonearena.com/news/Best-Android-browsers-2015-edition-design-features-and-performance_id67848 [Último acceso: Octubre 2015]

[11] Netcraft - Internet Security and Data Mining: *January 2015 Web Server Survey*. 15 de Enero 2015. http://news.netcraft.com/archives/2015/01/15/january-2015-web-server-survey.html [Último acceso: Julio 2015]

[12] The Internet Engineering Task Force (IETF): *The Transport Layer Security (TLS) Protocol Version 1.2*, rfc5246, Agosto 2008. http://tools.ietf.org/html/rfc5280

[13] The Internet Engineering Task Force (IETF): *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, rfc5280, Mayo 2008. http://tools.ietf.org/html/rfc5280

[14] Debian S.O Offcicial Web Site. Apache Web Server. Disponible en: https://wiki.debian.org/es/Apache [Último acceso: Octubre 2015]

[15] PowerTutor Official Web Site. University of Michigan. Disponible en: http://ziyang.eecs.umich.edu/projects/powertutor/ [Último acceso: Octubre 2015]

[16] Android Developers Official Web Site: *Android Studio Overview*. Disponible en: https://developer.android.com/tools/studio/index.html [Último acceso: Octubre 2015]

[17] RUSSELL HOLLY*: The 10 best Android browsers* (Mobile Nations ©, 1 Mayo 2015). Disponible en: http://www.androidcentral.com/10-best-android-browsers [Último acceso: Octubre 2015]

[18] Android Developers Official Web Site. SSLSocket. Disponible en: http://developer.android.com/reference/javax/net/ssl/SSLSocket.html [Último acceso: Octubre 2015]

[19] Paredes, David Maroto: *Estudio sobre seguridad en dispositivos móviles con sistema operativo Symbian*. Proyecto Fin de Carrera, Ingeniería Telemática, UC3M, Madrid, 2008.

[20] OpenSSL Offcial Web Site. OpenSSL Commands. Disponible en: https://www.openssl.org/docs/manmaster/apps/ [Último acceso: Octubre 2015]

[21] National Institute of Standards and Technology (NIST), *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations (Revision 1)*, NIST 800-52r1, April 2014. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r1.pdf [Último acceso: Octubre 2015]

APÉNDICE

Resultados completos

A.1 Pruebas con navegadores comerciales

- **NO Cipher**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,063158	0,149758	0,171332	0,108863	0,239525	0,12884
2ª repetición	0,077903	0,108667	0,117957	0,098188	0,204796	0,151348
3ª repetición	0,10783	0,083046	0,102167	0,096304	0,18755	0,115663
4ª repetición	0,066956	0,166806	0,111656	0,113157	0,237379	0,130916
5ª repetición	0,073795	0,10972	0,119684	0,108024	0,264526	0,140118
6ª repetición	0,079852	0,084547	0,109925	0,095459	0,285942	0,14281
7ª repetición	0,063791	0,251153	0,136615	0,079343	0,342046	0,111822
8ª repetición	0,105324	0,10675	0,09043	0,10144	0,170884	0,139756
9ª repetición	0,08301	0,087007	0,103047	0,098667	0,290892	0,147939
10ª repetición	0,067759	0,101936	0,096705	0,100738	0,284317	0,114327
Promedio	0,078938	0,124939	0,115952	0,100018	0,250786	0,132354
Varianza	0,015241	0,049478	0,022237	0,008858	0,05029	0,013666

Tabla A.1a. Tiempos de descarga ficheros 1K-10K sin cifrador

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,300985	1,365237	0,519771	2,430957	2,863689	2,516342
2ª repetición	0,287599	1,43281	0,356008	2,697246	3,291859	2,645386
3ª repetición	0,316996	1,147566	0,365874	2,539374	2,804227	2,724811
4ª repetición	0,288295	1,242234	0,31762	2,303266	2,79154	2,572186
5ª repetición	0,289023	1,181435	0,413456	2,359751	2,840911	2,461526
6ª repetición	0,287096	1,373165	0,377881	2,396199	2,831035	2,807821
7ª repetición	0,298122	1,482051	0,357229	2,400286	2,907057	2,791024
8ª repetición	0,30042	1,208105	0,475123	2,772707	2,844493	2,450315
9ª repetición	0,246821	1,453028	0,366269	2,680214	2,858128	2,843606
10ª repetición	0,290653	1,444814	0,339267	2,68403	3,39499	2,50129
Promedio	0,295465	1,3330445	0,38885	2,526403	2,9427929	2,631431
Varianza	0,009271	0,1195553	0,059965	0,1606415	0,2038731	0,143485

Tabla A.1b. Tiempos de descarga ficheros 100K-1M sin cifrador

- **AES128-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,127501	0,356596	0,291051	0,115964	0,360302	0,111723
2ª repetición	0,096618	0,458480	0,140903	0,119787	0,390650	0,096894
3ª repetición	0,102939	0,252342	0,142355	0,066754	0,374294	0,126039
4ª repetición	0,097054	0,268568	0,153283	0,083131	0,346866	0,095124
5ª repetición	0,085438	0,284246	0,177841	0,083086	0,448730	0,191270
6ª repetición	0,122051	0,225493	0,333350	0,132033	0,423895	0,254531
7ª repetición	0,137040	0,478231	0,157575	0,069779	0,367327	0,134433
8ª repetición	0,106923	0,241007	0,170443	0,128360	0,350256	0,212898
9ª repetición	0,152279	0,288599	0,283448	0,105920	0,419402	0,146801
10ª repetición	0,123008	0,246466	0,126541	0,137045	0,385566	0,111360
Promedio	0,115085	0,310003	0,197679	0,104186	0,386729	0,148107
Varianza	0,019723	0,086383	0,071092	0,025096	0,03234	0,051153

Tabla A.2a. Tiempos de descarga ficheros 1K-10K cifrador AES128-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,298331	1,700240	0,396877	2,690445	3,078661	2,919191
2ª repetición	0,281359	1,278206	0,387265	2,747106	2,940485	2,985178
3ª repetición	0,273496	1,746679	0,394816	2,573240	3,007940	2,282026
4ª repetición	0,310088	1,616204	0,528179	3,125760	2,997672	2,958344
5ª repetición	0,290563	1,567978	0,397119	2,684699	3,014765	2,959584
6ª repetición	0,278793	1,572638	0,354139	2,722762	2,992867	3,033944
7ª repetición	0,272492	1,796133	0,425104	2,576577	3,110753	2,888274
8ª repetición	0,290215	1,424997	0,404288	2,626911	3,055232	2,817086
9ª repetición	0,289008	1,385607	0,498813	2,621801	3,031565	3,094326
10ª repetición	0,298437	1,996077	0,369284	2,641227	3,124524	3,093881
Promedio	0,288278	1,6084759	0,415588	2,7010528	3,0354464	2,903183
Varianza	0,01138	0,2020812	0,052599	0,151834	0,0541496	0,222862

Tabla A.2b. Tiempos de descarga ficheros 100K-1M cifrador AES128-SHA

- **DES-CBC3-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,094062	0,346353	0,146821	0,105486	0,425164	0,197413
2ª repetición	0,086538	0,324569	0,187805	0,136325	0,518319	0,128913
3ª repetición	0,33194	0,261723	0,104605	0,073992	0,395887	0,12689
4ª repetición	0,074346	0,214778	0,132927	0,155817	0,377499	0,131775
5ª repetición	0,206954	0,28638	0,109283	0,104339	0,359209	0,12165
6ª repetición	0,13872	0,335238	0,163186	0,109365	0,323096	0,154705
7ª repetición	0,151256	0,259579	0,139672	0,090718	0,313323	0,094805
8ª repetición	0,154639	0,264127	0,139473	0,105154	0,288019	0,119223
9ª repetición	0,107801	0,305541	0,15795	0,098548	0,357394	0,127163
10ª repetición	0,113394	0,312826	0,119198	0,087283	0,411014	0,09547
Promedio	0,145965	0,291111	0,140092	0,106703	0,376892	0,129801
Varianza	0,072342	0,03903	0,02423	0,022529	0,062756	0,027917

Tabla A.3a. Tiempos de descarga ficheros 1K-10K cifrador DES-CBC3-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,277758	1,507726	0,388048	2,635574	4,038656	2,31978
2ª repetición	0,422329	1,082052	0,335281	2,673703	2,967966	2,291615
3ª repetición	0,308384	1,367319	0,477479	2,762704	2,954935	2,197928
4ª repetición	0,305225	1,629033	0,354148	2,619744	3,109015	2,296343
5ª repetición	0,292707	1,849064	0,327365	2,710226	3,054932	2,290852
6ª repetición	0,285486	1,527298	0,395863	2,713353	2,914963	2,278153
7ª repetición	0,279112	1,375698	0,335971	2,607063	2,958084	2,327088
8ª repetición	0,307124	1,320024	0,335787	2,718067	3,126818	2,260187
9ª repetición	0,309906	1,622385	0,362553	2,636152	3,089849	2,293424
10ª repetición	0,526848	1,630897	0,37096	2,988046	2,978398	2,282491
Promedio	0,331488	1,4911496	0,368346	2,7064632	3,1193616	2,283786
Varianza	0,076125	0,2026712	0,042636	0,1053997	0,3143094	0,033903

Tabla A.3b. Tiempos de descarga ficheros 100K-1M cifrador DES-CBC3-SHA

- **DHE-RSA-AES256-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,117095	0,342969	0,126282	0,073478	0,350225	0,529943
2ª repetición	0,105589	0,313889	0,206701	0,128436	0,348554	0,081689
3ª repetición	0,094135	0,253162	0,119585	0,102181	0,33475	0,087087
4ª repetición	0,09371	0,268981	0,106562	0,097031	0,349694	0,124548
5ª repetición	0,138131	0,218931	0,136853	0,086004	0,356203	0,114818
6ª repetición	0,089733	0,331915	0,170026	0,119085	0,365407	0,104751
7ª repetición	0,079188	0,238755	0,234944	0,106008	0,399414	0,130616
8ª repetición	0,137122	0,292627	0,203003	0,111202	0,40427	0,32667
9ª repetición	0,119986	0,235934	0,120511	0,110711	0,378018	0,148975
10ª repetición	0,138032	0,253369	0,110598	0,089777	0,34396	0,236889
Promedio	0,111272	0,275053	0,153507	0,102391	0,36305	0,188599
Varianza	0,020841	0,0408	0,044144	0,015522	0,022397	0,134594

Tabla A.4a. Tiempos de descarga ficheros 1K-10K cifrador DHE-RSA-AES256-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,298561	1,042094	0,282009	2,653007	2,945274	2,250012
2ª repetición	0,30355	1,024365	0,694283	2,714681	2,929222	2,791035
3ª repetición	0,279315	1,020646	0,34272	2,636421	2,982635	2,825293
4ª repetición	0,31683	1,047558	0,358098	2,640945	3,063557	2,948604
5ª repetición	1,311393	1,037673	0,343998	2,539305	2,963679	2,857548
6ª repetición	0,281831	1,349488	0,421179	2,764346	3,085954	3,011019
7ª repetición	0,285145	1,036463	0,365182	2,695439	2,958251	2,922375
8ª repetición	0,305539	1,109316	0,500685	2,517639	2,952324	2,840801
9ª repetición	0,274264	1,134186	0,343719	2,63844	2,952726	2,822438
10ª repetición	0,281308	1,085223	0,338064	3,004703	2,919954	2,977563
Promedio	0,393774	1,0887012	0,398994	2,6804926	2,9753576	2,824669
Varianza	0,306154	0,094066	0,112911	0,1288493	0,0525723	0,203892

Tabla A.4b. Tiempos de descarga ficheros 100K-1M cifrador DHE-RSA-AES256-SHA

- **ECDHE-ECDSA-AES128-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,129197	0,303321	0,187597	0,123772	0,355113	0,95075
2ª repetición	0,13597	0,41313	0,2317	0,091913	0,395442	0,100249
3ª repetición	0,146979	0,407102	0,360123	0,118132	0,458555	0,091949
4ª repetición	0,119029	0,516902	0,167651	0,099029	0,438606	0,094328
5ª repetición	0,132601	0,42527	0,124862	0,093659	0,375293	0,1688
6ª repetición	0,166703	0,314125	0,139121	0,103026	0,347008	0,119734
7ª repetición	0,106638	0,256666	0,194012	0,09674	0,498087	0,154278
8ª repetición	0,097449	0,308084	0,133086	0,089748	0,362957	0,133733
9ª repetición	0,128483	0,278299	0,16563	0,097451	0,363583	0,172723
10ª repetición	0,127888	0,273606	0,138311	0,131071	0,426539	0,134476
Promedio	0,129094	0,349651	0,184209	0,104454	0,402118	0,212102
Varianza	0,018446	0,081042	0,066486	0,01378	0,048324	0,247768

Tabla A.5a. Tiempos de descarga ficheros 1K-10K cifrador ECDHE-ECDSA-AES128-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,295631	1,287181	0,35318	2,689023	2,998999	2,918754
2ª repetición	0,307024	1,610695	0,432866	2,870577	2,982873	2,936562
3ª repetición	0,287911	1,879257	0,377582	2,533358	2,979226	2,848044
4ª repetición	0,302734	1,793893	0,391229	2,651016	3,092518	3,090999
5ª repetición	0,334123	1,629821	0,365309	2,624027	3,054596	2,440363
6ª repetición	0,306701	1,560778	0,41535	2,925082	3,038218	3,092556
7ª repetición	0,279452	1,549706	0,327452	2,59557	3,086163	2,936182
8ª repetición	0,277453	1,697789	0,379364	2,759762	2,951324	2,549537
9ª repetición	0,285546	1,554622	0,396639	2,75606	3,119257	3,002964
10ª repetición	0,316566	1,404607	0,460114	2,630318	3,056164	2,891092
Promedio	0,299314	1,5968349	0,389909	2,7034793	3,0359338	2,870705
Varianza	0,016854	0,1637701	0,03686	0,1175475	0,0529004	0,203825

Tabla A.5b Tiempos de descarga ficheros 100K-1M cifrador ECDHE-ECDSA-AES128-SHA

- **ECDHE-RSA-AES128-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,118172	0,348976	0,135797	0,135206	0,342588	0,103764
2ª repetición	0,035537	0,378507	0,126563	0,078974	0,399975	0,132476
3ª repetición	0,12202	0,280381	0,132345	0,174104	0,367701	0,141559
4ª repetición	0,080785	0,207555	0,674698	0,079664	0,406448	0,106925
5ª repetición	0,113835	0,24729	0,136151	0,266659	0,365084	0,096679
6ª repetición	0,111195	0,309517	0,114091	0,106162	0,343453	0,158657
7ª repetición	0,151753	0,325925	0,148569	0,081238	0,407046	0,113636
8ª repetición	0,0999	0,219932	0,187349	0,11258	0,380025	0,127268
9ª repetición	0,110009	0,19984	0,207027	0,108695	0,387354	0,092668
10ª repetición	0,135471	0,281994	0,16626	0,10591	0,381922	0,184478
Promedio	0,107868	0,279992	0,202885	0,124919	0,37816	0,125811
Varianza	0,030124	0,058136	0,159624	0,054599	0,022278	0,027845

Tabla A.6a. Tiempos de descarga ficheros 1K-10K cifrador ECDHE-RSA-AES128-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,282623	1,620447	1,515278	2,795669	3,016528	3,082406
2ª repetición	0,296386	1,841913	0,453453	2,661823	3,10376	2,823075
3ª repetición	0,282766	1,393374	0,449395	2,609196	3,447203	3,018417
4ª repetición	0,278429	1,371819	0,365168	2,665452	2,924411	2,836136
5ª repetición	0,284852	1,533773	0,36819	2,632941	3,079817	2,882281
6ª repetición	0,282708	1,567209	0,422527	2,790859	2,979288	2,37136
7ª repetición	0,443347	1,408737	0,378901	2,635807	3,016321	2,835556
8ª repetición	0,281214	1,588489	0,412752	2,65236	3,065956	3,000275
9ª repetición	0,282499	3,09372	0,373414	2,67591	3,093663	2,783186
10ª repetición	0,308693	1,805881	0,333199	2,666067	3,164055	2,869391
Promedio	0,302352	1,7225362	0,507228	2,6786084	3,0891002	2,850208
Varianza	0,047785	0,4815923	0,338035	0,0602887	0,1358154	0,184679

Tabla A.6b. Tiempos de descarga ficheros 100K-1M cifrador ECDHE-RSA-AES128-SHA

- **AES256-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,13852	0,233803	0,1763	0,122359	0,284651	0,09313
2ª repetición	0,108902	0,272769	0,121516	0,100815	0,363585	0,130576
3ª repetición	0,116624	0,277966	0,182124	0,076863	0,367285	0,107821
4ª repetición	0,145032	0,368705	0,130687	0,068452	0,375176	0,246187
5ª repetición	0,11581	0,206212	0,116502	0,096464	0,405938	0,122715
6ª repetición	0,152907	0,256113	0,121894	0,087075	0,341291	0,690632
7ª repetición	0,08688	0,258578	0,1608	0,105202	0,336499	0,110809
8ª repetición	0,122646	0,23742	0,09987	0,106445	0,361874	0,167636
9ª repetición	0,102102	0,324465	0,13852	0,14859	0,35021	0,198583
10ª repetición	0,116425	0,392159	0,138099	0,080522	0,353166	0,196209
Promedio	0,120585	0,282819	0,138631	0,099279	0,353968	0,20643
Varianza	0,019074	0,057292	0,025359	0,022393	0,029612	0,167979

Tabla A.7a. Tiempos de descarga ficheros 1K-10K cifrador AES256-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,275756	0,518035	1,546612	2,266804	2,955739	2,37593
2ª repetición	0,418374	1,4947	0,34231	2,293947	3,059953	2,768266
3ª repetición	0,331165	1,423653	0,353379	2,324196	3,001104	2,832863
4ª repetición	0,301075	1,647338	0,368807	2,31448	2,932689	2,81077
5ª repetición	0,307188	1,48102	0,361872	2,307566	3,036312	2,793157
6ª repetición	0,311965	1,55125	0,361247	2,309657	2,965943	2,774894
7ª repetición	0,340162	1,915879	0,409087	2,398359	2,97112	2,955733
8ª repetición	0,348582	1,684257	0,300393	2,332104	2,933375	2,787084
9ª repetición	0,31735	1,605233	0,387308	2,341199	3,011094	3,042999
10ª repetición	0,297738	1,369649	0,359862	2,321085	2,940325	3,042671
Promedio	0,324936	1,469101	0,479088	2,32094	2,980765	2,818437
Varianza	0,037187	0,349318	0,356844	0,032476	0,042215	0,179353

Tabla A.7b. Tiempos de descarga ficheros 100K-1M cifrador AES256-SHA

- **DHE-RSA-AES128-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,096431	0,344267	0,148136	0,283492	0,354285	0,109816
2ª repetición	0,117693	0,350687	0,145943	0,111568	0,368029	0,140088
3ª repetición	0,13814	0,230516	0,173541	0,087136	0,340303	0,088763
4ª repetición	0,125317	0,22462	0,100627	0,129166	0,320929	0,120924
5ª repetición	0,123605	0,282113	0,103861	0,143023	0,50405	0,104323
6ª repetición	0,076221	0,267733	0,163172	0,070262	0,432141	0,165635
7ª repetición	0,137137	0,228122	0,130961	0,107993	0,426611	0,106903
8ª repetición	0,09993	0,289056	0,153023	0,122654	0,411422	0,262361
9ª repetición	0,086516	0,32547	0,144547	0,146137	0,355039	0,130198
10ª repetición	0,116148	0,194129	0,136488	0,139553	0,338998	0,105505
Promedio	0,111714	0,273671	0,14003	0,134098	0,385181	0,133452
Varianza	0,019984	0,051586	0,022145	0,054952	0,054012	0,047702

Tabla A.8a. Tiempos de descarga ficheros 1K-10K cifrador DHE-RSA-AES128-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,283735	1,375895	0,425162	2,669467	3,073427	2,97758
2ª repetición	0,285659	1,515535	0,432358	2,660515	2,959035	3,063312
3ª repetición	0,296381	1,285196	0,340744	2,890827	2,988687	2,256238
4ª repetición	0,30772	1,10741	0,485846	2,646102	2,970582	2,961049
5ª repetición	0,305194	1,475453	0,380077	2,707273	2,979398	2,995251
6ª repetición	0,272057	1,135099	0,581331	2,585385	2,966663	3,071676
7ª repetición	0,283968	1,331461	0,467869	2,779897	2,91956	2,953534
8ª repetición	0,318039	1,194047	2,033196	2,47729	2,936926	2,838127
9ª repetición	0,272851	1,480761	0,526334	2,514313	2,926183	2,893313
10ª repetición	0,285276	1,118219	0,355223	2,768335	2,985147	2,948681
Promedio	0,291088	1,301908	0,602814	2,66994	2,970561	2,895876
Varianza	0,014466	0,150133	0,482121	0,118785	0,041292	0,223233

Tabla A.8b. Tiempos de descarga ficheros 100K-1M cifrador DHE-RSA-AES128-SHA

- **ECDHE-ECDSA-AES128-GCM-SHA256**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,112711	0,378486	0,214353	0,07761	0,366892	0,111998
2ª repetición	0,138693	0,298791	0,128237	0,086688	0,448672	0,111182
3ª repetición	0,128167	0,322831	0,154739	0,077462	0,367259	0,140139
4ª repetición	0,124142	0,3526	0,149472	0,182584	0,396479	0,144235
5ª repetición	0,09556	0,386139	0,121837	0,092807	0,34272	0,137067
6ª repetición	0,103945	0,332204	0,168149	0,173729	0,332088	0,114247
7ª repetición	0,112539	0,248758	0,124586	0,126032	0,369879	0,183225
8ª repetición	0,094761	0,340559	0,148009	0,090369	0,382646	0,119533
9ª repetición	0,123293	0,21883	0,176969	0,100569	0,333604	0,131591
10ª repetición	0,114603	0,229797	0,178484	0,119748	0,550762	0,119031
Promedio	0,114841	0,3109	0,156484	0,11276	0,3891	0,131225
Varianza	0,013446	0,057014	0,027438	0,036093	0,062929	0,020814

Tabla B.9a Tiempos de descarga ficheros 1K-10K cifrador ECDHE-ECDSA-AES128-GCM-SHA256

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,281913	1,629107	0,426673	2,675462	3,040988	2,868065
2ª repetición	0,294412	1,67492	0,375469	2,809251	3,019976	2,84219
3ª repetición	0,286611	1,892953	0,385495	2,633514	3,03587	2,906875
4ª repetición	0,277995	1,779178	0,408513	2,642758	2,977658	2,96644
5ª repetición	0,295557	1,761113	0,537765	2,655763	3,153474	3,110692
6ª repetición	0,31173	1,474548	0,331059	2,828299	3,005848	2,856246
7ª repetición	0,276235	1,907337	0,386996	2,642325	3,004564	2,820015
8ª repetición	0,276822	1,625179	0,422993	3,251879	3,020623	2,907415
9ª repetición	0,282388	1,642854	0,431417	2,759858	3,104927	3,047456
10ª repetición	0,284543	1,558906	0,404247	2,635848	3,045981	2,412552
Promedio	0,286821	1,69461	0,411063	2,753496	3,040991	2,873795
Varianza	0,010456	0,132358	0,050801	0,180452	0,049215	0,177218

Tabla A.9b. Tiempos de descarga ficheros 100K-1M cifrador ECDHE-ECDSA-AES128-GCM-SHA256

- **ECDHE-ECDSA-AES256-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,050071	0,219622	0,133239	0,079717	0,329291	0,106389
2ª repetición	0,098454	0,360719	0,131521	0,129811	0,359112	0,164591
3ª repetición	0,141026	0,240917	0,176823	0,244307	0,367835	0,095067
4ª repetición	0,145438	0,265787	0,136365	0,117387	0,37927	0,142404
5ª repetición	0,168659	0,238162	0,218465	0,188643	0,404505	0,301038
6ª repetición	0,103915	0,240875	0,174828	0,102803	0,365931	0,101642
7ª repetición	0,095452	0,2104	0,144461	0,078357	0,370629	0,166398
8ª repetición	0,12345	0,417924	0,1787	0,102531	0,337512	0,183359
9ª repetición	0,122626	0,250895	0,138118	0,125353	0,39971	0,1438
10ª repetición	0,03888	0,218867	0,126392	0,099116	0,360736	0,134926
Promedio	0,108797	0,266417	0,155891	0,126803	0,367453	0,153961
Varianza	0,038751	0,064636	0,02835	0,049221	0,022434	0,056419

Tabla A.10a. Tiempos de descarga ficheros 1K-10K cifrador ECDHE-ECDSA-AES256-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,3085	1,21292	0,415611	2,685467	2,960572	2,881548
2ª repetición	0,281564	1,802173	0,388264	2,767387	2,98758	2,835148
3ª repetición	0,27801	1,670759	0,532954	2,677708	3,045644	2,976183
4ª repetición	0,280442	1,671575	1,058943	2,606823	3,197583	2,950676
5ª repetición	0,294507	1,731277	0,356261	2,613227	3,109127	2,888567
6ª repetición	0,296844	1,566561	0,420665	2,637419	3,042492	3,143883
7ª repetición	0,422947	1,911022	0,473369	2,714677	3,205528	2,795185
8ª repetición	0,284084	1,793188	0,409544	2,65268	3,069565	2,977652
9ª repetición	0,311766	1,376633	0,387692	2,946781	3,149392	2,423723
10ª repetición	0,286324	1,540707	0,390409	2,598278	3,174284	2,951888
Promedio	0,304499	1,627682	0,483371	2,690045	3,094177	2,882445
Varianza	0,041001	0,199747	0,19769	0,09904	0,082236	0,177762

Tabla A.10b. Tiempos de descarga ficheros 100K-1M cifrador ECDHE-ECDSA-AES256-SHA

- **ECDHE-RSA-AES256-SHA**

TIEMPO DESCARGA FICHERO (segundos)						
	1K			10K		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,070932	0,242322	0,120818	0,079013	0,443059	0,188414
2ª repetición	0,084567	0,244867	0,137979	0,110188	0,352964	0,102472
3ª repetición	0,084243	0,270903	0,131737	0,074081	0,357527	0,107075
4ª repetición	0,067183	0,290073	0,16888	0,089622	0,354965	0,120913
5ª repetición	0,090266	0,299723	0,131753	0,109884	0,422515	0,145699
6ª repetición	0,090275	0,244391	0,107569	0,077148	0,339365	0,261553
7ª repetición	0,101054	0,209795	0,099429	0,083709	0,384914	0,106207
8ª repetición	0,10188	0,200572	0,140218	0,08185	0,344166	0,10479
9ª repetición	0,121187	0,248712	0,123228	0,105034	0,382249	0,093705
10ª repetición	0,088201	0,295919	0,114314	0,080767	0,358129	0,103322
Promedio	0,089979	0,254728	0,127593	0,08913	0,373985	0,133415
Varianza	0,014786	0,032524	0,018552	0,013237	0,032753	0,050443

Tabla A.11a. Tiempos de descarga ficheros 1K-10K cifrador ECDHE-RSA-AES256-SHA

TIEMPO DESCARGA FICHERO (segundos)						
	100K			1M		
	CHROME	FIREFOX	OPERA	CHROME	FIREFOX	OPERA
1ª repetición	0,368924	1,181386	0,354334	2,724751	2,520214	2,212446
2ª repetición	0,309187	1,859541	0,363234	2,836956	2,944095	2,223665
3ª repetición	0,317603	1,411133	1,073858	2,550316	2,991264	2,372973
4ª repetición	0,307842	1,721907	0,404324	2,654722	2,982935	2,22895
5ª repetición	0,282962	1,541513	0,335261	2,89122	3,040986	2,340611
6ª repetición	1,131849	1,145526	0,353647	2,757665	2,990186	2,348171
7ª repetición	0,306413	1,640623	0,342812	2,686902	3,071516	2,309587
8ª repetición	0,276438	1,617367	0,519348	2,614446	2,954204	2,329854
9ª repetición	0,279363	1,731235	0,571879	2,623432	2,988274	2,187277
10ª repetición	0,284291	1,801774	0,390213	2,694839	2,995826	2,291294
Promedio	0,386487	1,565201	0,470891	2,703525	2,94795	2,284483
Varianza	0,249787	0,234707	0,214657	0,098474	0,14688	0,062604

Tabla A.11b. Tiempos de descarga ficheros 100K-1M cifrador ECDHE-RSA-AES256-SHA

A.2 Pruebas con navegador programático

- **Resultados prueba de tiempos**

TIEMPOS RSA (segundos)					
	RSA1024	**RSA2048**	**RSA3072**	**RSA7680**	**RSA15360**
1ª repetición	42	48	51	205	1062
2ª repetición	35	45	51	207	1092
3ª repetición	35	49	60	217	1032
4ª repetición	34	46	57	206	1025
5ª repetición	38	43	51	217	1030
6ª repetición	35	46	52	188	1064
7ª repetición	36	47	53	185	1037
8ª repetición	39	44	52	204	1043
9ª repetición	38	41	49	182	1066
10ª repetición	34	44	52	200	1023
Promedio	**36.6**	**45.3**	**52.8**	**201.1**	**1047.4**
Varianza	**2.45764115**	**2.28254244**	**3.09192497**	**11.7681774**	**21.42055088**

Tabla A.12. Tiempos handshake certificados RSA

TIEMPOS ECDSA (segundos)					
	ECC160	**ECC224**	**ECC256**	**ECC384**	**ECC512**
1ª repetición	42	50	47	67	93
2ª repetición	39	52	47	67	95
3ª repetición	42	46	55	66	96
4ª repetición	39	47	51	64	100
5ª repetición	40	44	47	64	101
6ª repetición	39	45	48	66	94
7ª repetición	38	53	56	63	95
8ª repetición	38	43	48	65	94
9ª repetición	42	44	52	64	97
10ª repetición	42	45	46	64	97
Promedio	**40.1**	**46.9**	**49.7**	**65**	**96.2**
Varianza	**1.64012195**	**3.36005952**	**3.40734501**	**1.34164079**	**2.48193473**

Tabla A.13. Tiempos handshake certificados ECDSA

- **Resultados prueba de energía**

ENERGÍA RSA (miliJulios)					
	RSA1024	**RSA2048**	**RSA3072**	**RSA7680**	**RSA15360**
1ª repetición	20	26	42	27	152
2ª repetición	26	27	72	52	165
3ª repetición	24	26	63	97	65
4ª repetición	19	29	67	75	54
5ª repetición	31	32	42	43	195
6ª repetición	27	52	34	52	75
7ª repetición	36	38	24	40	137
8ª repetición	52	59	63	84	52
9ª repetición	27	29	30	52	118
10ª repetición	17	31	29	91	123
Promedio	**27.9**	**34.9**	**46.6**	**61.3**	**113.6**
Varianza	**9.67935948**	**10.9402925**	**17.0188131**	**22.5479489**	**47.53567082**

Tabla A.14. Energía empleada en handshake certificados RSA

ENERGÍA ECDSA (miliJulios)					
	ECC160	**ECC224**	**ECC256**	**ECC384**	**ECC512**
1ª repetición	37	71	29	95	59
2ª repetición	41	33	54	94	44
3ª repetición	27	29	33	71	58
4ª repetición	59	56	74	32	47
5ª repetición	39	36	26	35	40
6ª repetición	22	90	29	91	40
7ª repetición	51	26	64	84	97
8ª repetición	56	30	68	110	34
9ª repetición	46	71	64	31	39
10ª repetición	60	72	31	46	62
Promedio	**43.8**	**51.4**	**47.2**	**68.9**	**52**
Varianza	**12.3838605**	**22.1006787**	**18.2690996**	**28.6232423**	**17.5499288**

Tabla A.15. Energía empleada en handshake certificados ECC